Homework 1
Ruby calisthenics

Daniel Lincoln, 202042829

Departamento de Ciências da Computação – Universidade de Brasília (UnB)
CIC0105 - Engenharia de Software
2 de maio de 2025

Resumo. *O relatório apresenta as soluções dos exercícios das Partes 01 a 07 do Homework 01, com explicações dos algoritmos usados. São discutidas alternativas de implementação em Ruby, comparando estilos e estruturas da linguagem. Inclui também uma análise do tempo e esforço dedicados, além de um breve feedback sobre a experiência.*

1. Introdução

1.1. Objetivos

Analisar e desenvolver soluções em Ruby para os desafios propostos, explorando diferentes abordagens e recursos da linguagem.

1.2. Resultados obtidos

A estrutura dos testes do código utiliza .all? e verificações explícitas para igualdade com resultados esperados e tratamento de exceções. Considerando que as implementações dos métodos (palindrome?, count_words, etc.) estão corretas para passar nos testes definidos, os resultados para as partes 1 a 7, conforme o código está estruturado para reportar, é 'success' para todos os sub-itens (a, b, c, etc.).

1.3. Principais Estruturas Utilizadas

O código utiliza diversas estruturas e técnicas comuns em Ruby:

Classes e Objetos: Estrutura básica do código, com classes como Hmw1, Dessert, JellyBean, Foo, Foo2, CurrencyWrapper, CartesianProduct. Objetos são instanciados para representar sobremesas, objetos com histórico de atributos, envoltórios de moeda e produtos cartesianos.

Métodos de Classe (self.method_name): Utilizados na classe Hmw1 para organizar a execução e os testes das diferentes partes. Também em Hmw1 para métodos utilitários como palindrome?, count_words, rps_game_winner, rps_tournament_winner, combine_anagrams.

Variáveis de Instância (@variable): Utilizadas para armazenar o estado dos objetos, como @name, @calories, @flavor em Dessert/JellyBean. @bar em Foo/Foo2, @value, @currency em CurrencyWrapper, e @seq1, @seq2 em CartesianProduct.

Variáveis de Classe (@@variable): Utilizadas em Numeric para armazenar as taxas de câmbio de forma compartilhada entre todas as instâncias da classe e seus descendentes. @@currencies armazena um Hash mapeando nomes de moeda para seus valores em dólares.

Herança (<Superclass): A classe JellyBean herda de Dessert, reutilizando a lógica de healthy? e estendendo/sobrescrevendo delicious?.

Mixins (include Module): A classe CartesianProduct inclui o mixin Enumerable. Isso permite que a classe utilize métodos de iteração poderosos como map, select, reduce, etc., desde que implemente o método each.

Tratamento de Exceções (begin...rescue...end, raise): Utilizado em rps_game_winner para sinalizar erros com o número de jogadores ou estratégias inválidas, e testado em p2a. CurrencyWrapper#in também levanta ArgumentError para moedas desconhecidas. p6a testa o rescue ArgumentError para conversão de moeda.

Métodos Dinâmicos (method_missing, define_method, class_eval):

method_missing: Utilizado em Numeric e CurrencyWrapper para interceptar chamadas de métodos que não existem explicitamente, como 3.yen ou 7.euros. Em Numeric, ele cria um CurrencyWrapper. Em CurrencyWrapper, ele tenta converter para a moeda solicitada ou delega a chamada para o valor numérico subjacente.

define_method e class_eval: Utilizados no método attr_accessor_with_history adicionado à classe Class. class_eval executa código no contexto da classe que está sendo modificada. define_method cria dinamicamente os métodos getter (attr_name) e setter (attr_name=) e o método para acessar o histórico (attr_name_history).

Manipulação de Strings e Coleções: Uso extensivo de métodos como gsub, downcase, scan, chars, sort, each_with_object, group_by, zip, all? para processar dados, verificar condições e agrupar elementos.

Blocos e yield: Utilizados na implementação de CartesianProduct#each para passar cada par gerado para o bloco fornecido pelo iterador. enum_for(:each) é usado para retornar um Enumerator quando nenhum bloco é fornecido. each_with_object em count_words também utiliza um bloco

1.4. Principais Algoritmos/Lógica

Verificação de Palíndromo: Para strings, remove caracteres não alfanuméricos, converte para minúsculas e compara com a string revertida. Para enumeráveis, simplesmente compara o com o revertido.

Contagem de Palavras: Converte a string para minúsculas, usa expressões regulares (\b\w+\b) para encontrar todas as palavras, e então itera sobre elas usando each_with_object com um Hash inicializado com zero para contar a frequência de cada palavra.

Vencedor de RPS: Verifica o número de jogadores e a validade das estratégias (R, P, S). Usa um mapa de ranks para calcular o vencedor baseado na diferença modular 3 dos ranks das estratégias.

Vencedor de Torneio RPS: Implementa uma lógica recursiva: se a entrada é um torneio aninhado (uma array de arrays de arrays), resolve recursivamente os sub-torneios e joga os vencedores uns contra os outros; se é um único jogo (uma array de arrays), resolve o jogo.

Agrupar Anagramas: Normaliza cada palavra (minúsculas, caracteres ordenados)

e usa group_by para agrupar as palavras que têm a mesma forma normalizada. O group_by retorna um Hash onde a chave é a forma normalizada e o valor é uma array de palavras originais que correspondem a essa chave. .values extrai apenas as arrays de palavras (os grupos de anagramas).

Histórico de Atributos: A lógica em attr_accessor_with_history define um método setter que, antes de atribuir o novo valor ao atributo, adiciona o valor à uma array de histórico associada à instância (@attr_name_history). Um método getter para o histórico é também definido dinamicamente.

Conversão de Moeda: Utiliza method_missing e in para criar um envoltório (CurrencyWrapper) quando um número é chamado com um nome de moeda. O CurrencyWrapper armazena o valor e a moeda original. O método in no CurrencyWrapper calcula o valor convertido usando as taxas armazenadas em @@currencies e retorna um novo CurrencyWrapper com o valor convertido e a nova moeda. method_missing em CurrencyWrapper também permite operações no valor subjacente, mantendo a moeda.

Produto Cartesiano: O método each na classe CartesianProduct itera sobre cada elemento da primeira sequência e, para cada um, itera sobre cada elemento da segunda sequência, gerando um par [elemento1, elemento2] e passando-o para o bloco fornecido.

2. Propostas de Implementações Alternativas e Análise

2.1. Paradigma Funcional vs. Imperativo

O código atual apresenta uma mistura. combine_anagrams com group_by é um exemplo de estilo funcional, conciso. count_words com each_with_object também se inclina para o funcional. Por outro lado, rps_game_winner e a manipulação de variáveis de instância em attr_accessor_with_history são mais imperativas.

2.1.1. Alternativas

Mais funcional: Poderia-se reescrever rps_game_winner usando casamento de padrões ou estruturas de dados que representem as regras de forma mais declarativa, embora a implementação atual com ranks e módulo 3 seja eficiente. palindrome? para coleções poderia usar zip com reverse e all? de forma mais explícita, embora já esteja razoavelmente funcional. Processamento de listas (como em p1a, p1b, p6c, p7a) já faz uso extensivo de .all? e zip, que são funcionais.

Vantagens do funcional: Código geralmente mais conciso, menos estado mutável explícito, mais fácil de testar unidades independentes, potencial para paralelização.

Desvantagens do funcional: Pode ser menos intuitivo para iniciantes, certas operações (como I/O ou manipulação de estado complexo) são mais naturalmente expressas de forma imperativa.

Mais imperativo: Reescrever combine_anagrams com loops aninhados explícitos para comparar cada par de palavras seria uma abordagem mais imperativa, mas significativamente menos eficiente e mais verbosa do que group_by.

Vantagens do imperativo: Controle direto sobre o fluxo de execução e gerencia-

mento de memória, mais familiar para muitos programadores, bom para tarefas que envolvem mudanças de estado sequenciais.

Desvantagens do imperativo: Pode levar a código mais longo e difícil de manter, mais propenso a efeitos colaterais inesperados.

2.2. Uso de Outras Coleções Ruby

O código atual usa Array e Hash. O Hash para contagem de palavras é padrão e eficiente, e Arrays para listas de anagramas e para a estrutura de torneio RPS é apropriado pela natureza sequencial ou aninhada dos dados. Alternativas para o torneio (como grafos) seriam excessivamente complexas para este problema.

2.2.1. Alternativas

Struct ou classes customizadas: Em vez de representar jogadores em RPS como arrays ['Nome', 'Estrategia'], poderia ser criada uma Struct Player ou uma classe Player com atributos name e strategy.

Vantagens: Melhora a clareza (nomes significativos em vez de índices), potencial para adicionar comportamento aos objetos de dados, mais robusto a erros (menos propenso a acessar um índice inexistente).

Desvantagens: Mais verbosidade na definição.

2.3. Uso de Mixins do Ruby

O código atual usa Enumerable em CartesianProduct. Sem Mixins, para CartesianProduct, o método each ainda seria necessário para permitir a iteração básica (ex: c.each ...), mas métodos como map, select, to_a não estariam disponíveis diretamente na instância de CartesianProduct. Seria necessário converter explicitamente para um Array primeiro (c.to_a.map ...).

Vantagens de usar mixins: Adiciona um conjunto rico de métodos úteis com a implementação de apenas um ou poucos métodos requeridos pelo mixin (como each para Enumerable). Promove reuso de código e conformidade com interfaces comuns.

Desvantagens de usar mixins (ou não usar): Não usar mixins significa reinventar a roda ou ter que converter objetos para tipos que possuem esses métodos, tornando o código mais verboso. O uso excessivo de mixins pode potencialmente poluir o namespace de uma classe, embora em Ruby isso raramente seja um problema sério.

2.3.1. Outros Mixins Potenciais

Comparable: Poderia ser útil se fosse necessário comparar CurrencyWrappers ou Desserts com base em valor/calorias, implementando o método <=>.

2.4. Yield

O código atual usa yield em CartesianProduct#each. Sem yield diretamente para iteração, o método each poderia retornar a array completa do produto cartesiano de uma vez (return result_array).

Vantagens de usar yield: Eficiência de memória ao trabalhar com coleções potencialmente grandes, pois os elementos são gerados um por um e não precisam ser armazenados na memória simultaneamente em uma array intermediária completa. Permite a criação de iteradores customizados que se integram bem com os métodos de Enumerable.

Desvantagens de usar yield: O código que utiliza yield pode ser um pouco menos direto de entender para quem não está familiarizado com blocos e iteradores em Ruby.

Alternativa a yield que mantém a ”lazy evaluation”: Retornar um Enumerator explicitamente. Em CartesianProduct#each, return enum_for(:each) já faz isso quando nenhum bloco é dado. Poderia-se sempre retornar um Enumerator e deixar o usuário chamar .each ou outros métodos nele.

3. Análise de Tempo e Esforço

O tempo e esforço para redigir este código dependeram do nível de familiaridade com Ruby e seus conceitos avançados. Conceitos básicos (classes, métodos, variáveis, condicionais, loops) foram diretos, enquanto manipulação de coleções (Array, Hash) com métodos idiomáticos de Ruby (map, select, group_by, each_with_object, all?, zip) tornaram necessária alguma pesquisa na biblioteca padrão.

Recursão (em rps_tournament_winner) é um conceito fundamental, nesse caso específico não exigindo muito tempo para imaginar a lógica, mas em contextos com casos base mais complexos poderia ter tomado mais tempo.

O uso de include Enumerable e a implementação de each em CartesianProduct exigiram a compreensão do protocolo de iteração.

As partes mais desafiadoras para entender foram as extensões de classes built-in (Numeric, Class, String), principalmente na forma como foram utilizados method_missing, e class_eval. Esses são recursos poderosos, mas menos comuns e custaram a serem compreendidos de acordo com o modelo de objetos dinâmicos do Ruby.

No geral, entender as partes 1, 2a, 3, 4 foi relativamente rápido. As partes 2b, 5, 6 (principalmente) e 7 exigiram mais tempo e pesquisa sobre os recursos específicos usados.

Métodos como palindrome?, count_words, rps_game_winner, combine_anagrams, e as classes Dessert/JellyBean foram implementados com esforço moderado voltado mais para a sintaxe do que a lógica. A recursão do torneio RPS exigiu mais tempo para garantir que as condições de base e passo recursivo estivessem corretas.

attr_accessor_with_history e a funcionalidade de moedas com method_missing e extensão de classes foram implementações mais avançadas que exigiram um esforço considerável, incluindo pesquisa e testes cuidadosos para garantir que se integrassem corretamente com o comportamento existente.

CartesianProduct exigiu a compreensão de como implementar um iterador e a inclusão de Enumerable.

4. Análise das Respostas

O código desenvolvido já inclui um conjunto de testes. A análise crítica envolveu:

Revisar a lógica dos métodos para garantir que lidam com casos de borda (entradas vazias, entradas inválidas, etc.).

Verificar a eficiência dos algoritmos (complexidade de tempo e espaço). combine_anagrams usando group_by é eficiente. count_words também é eficiente. A recursão do torneio RPS parece direta, mas poderia ser otimizada para torneios muito grandes. O produto cartesiano em each gera pares um por um, o que é bom para memória.

Garantir que as extensões de classe (Numeric, Class, String) não causassem conflitos inesperados com outras bibliotecas. A forma como method_missing é usada, delegando respond_to? e public_send minimiza conflitos.

5. Conclusão

A experiência de trabalhar com este conjunto de exercícios foi bastante enriquecedora para a aprendizagem de Ruby, abrangendo uma boa variedade de tópicos. Pontos positivos foram: cobrir um espectro amplo de funcionalidades e paradigmas em Ruby; desenvolver uma estrutura de teste que ajudou a verificar a correção das implementações; lidar com attr_accessor_with_history e as extensões de moeda, ótimas introduções ao poder da meta-programação em Ruby; implementar CartesianProduct com Enumerable, demonstrando o padrão de iterador e o benefício dos mixins. No geral, é um conjunto desafiador e gratificante de exercícios que contribui o aprendizado de vários aspectos importantes e poderosos da linguagem Ruby.