UNIVERSIDADE FEDERAL DE MINAS GERAIS
Departamento de Ciência da Computação

TRABALHO PRÁTICO I

Simulação de Batalha de Pokémon

Lívia Caroline Rodrigues Pereira

Junho de 2024

SUMÁRIO

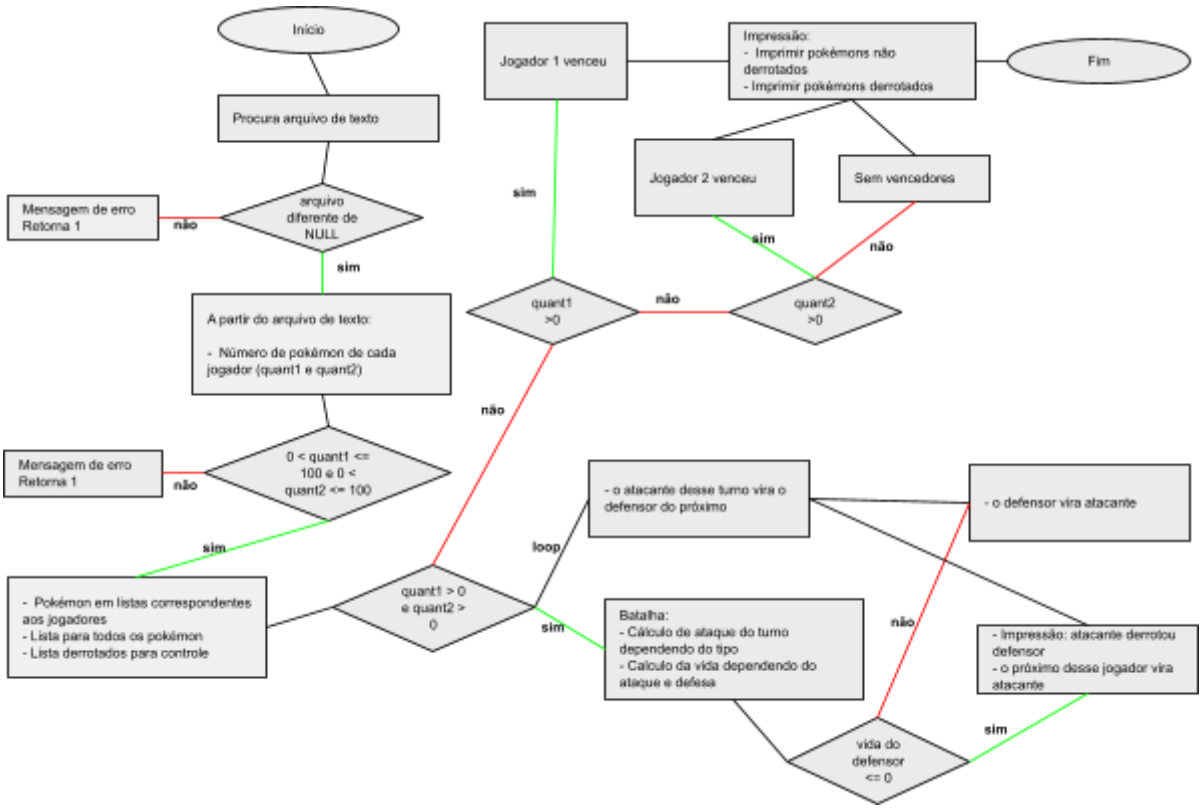
1. Introdução

Esta documentação explica de forma concisa as decisões e procedimentos utilizados na construção do código desenvolvido para o Trabalho Prático da matéria de Programação e Desenvolvimento de Software I, no 1º Período do curso de Ciência de Dados. O trabalho consiste em um programa de simulação de batalhas Pokémon entre dois jogadores, em que os turnos se alternam até que um dos jogadores fique sem Pokémon. O ataque de cada turno, dependendo dos tipos dos Pokémon, pode aumentar ou diminuir e, se for maior que a defesa, a diferença é subtraída da vida do defensor; caso contrário, apenas 1 ponto de vida é retirado. O código recebe um arquivo .txt com as informações dos Pokémon de ambos os jogadores e imprime o resultado de cada batalha, o vencedor e seus sobreviventes, além dos Pokémon derrotados. O objetivo é desenvolver habilidades de programação e facilitar o cálculo de batalhas Pokémon com muitos participantes.

2. Descrição do Algoritmo e Procedimentos utilizados

O código em C é um simulador de batalhas Pokémon que lê dados de um arquivo para determinar o vencedor entre dois jogadores, cada um com até 100 Pokémon. Utiliza as funções *fgets* e *strtok* para ler e tokenizar os dados de entrada, validando-os para garantir que não sejam negativos e convertendo os tipos dos Pokémon para minúsculas. A lógica de combate é baseada em turnos, organizados por laços de repetição, onde o atacante causa dano ao defensor, considerando vantagens e desvantagens dos tipos, e a vida do defensor é atualizada conforme o dano recebido. O programa utiliza alocação dinâmica de memória com *malloc* e *free* para armazenar as listas de Pokémon e usa estruturas (*struct*) para representar os Pokémon. A função *gravacao* organiza a leitura e armazenamento dos dados em arrays de structs. Durante a batalha, o controle do atacante e defensor alterna a cada turno. Após as batalhas, o programa imprime o vencedor, a lista de sobreviventes e os Pokémon derrotados, mostrando a eficiência do uso de diversas funções na linguagem C.

a. Diagrama lógico



b. Documentação esperada

O programa espera receber um arquivo de texto contendo as informações dos pokémon dos dois jogadores participantes. No caso, o arquivo *.txt* deve ser nomeado *pokemon.txt*. Além disso, o programa conta com alguns comentários que, caso sejam inseridos no código, permitem que o usuário entre com o nome do arquivo, por solicitação no terminal ou a partir do *stdin* e *stdout*. A primeira linha do programa deve receber o número de pokémon de cada jogador, seguida por linhas consecutivas dos respectivos pokémon. Por exemplo, se a primeira linha estabelece 2 3, as duas próximas linhas dizem respeito à pokémon do primeiro jogador e as 3 subsequentes, a pokémon do segundo jogador. Nas linhas dos pokémon, a ordem deve ser: Nome, Poder de ataque, Poder de defesa, Vida inicial e Tipo.

c. Execução

Input: 1 1

Processo:

1. Leitura do número de Pokémon para cada jogador.

2. Criação de duas listas, do tipo Poke, separando os pokémon de cada um dos jogadores.

3. A partir da função *gravacao*, gravação dos respectivos pokémon nas listas dos jogadores. A função recebe a lista correspondente e a quantidade de pokémon que será armazenada, além do arquivo a ser lido. A partir disso, internamente cria uma variável do tipo poke, coleta os atributos e, ao fim, adiciona a variável a um índice na lista, até que o tamanho informado seja ocupado.

4. Adicionamos na lista *todos* os pokémon dos dois jogadores, calculando também o total de pokémon no jogo. Uma lista de valores booleanos chamada *derrotados* também é criada para controle dos pokémon derrotados (a princípio todos os índices são 0).

Input: Squirtle 15 25 40 agua

Input: Charmander 25 12 36 fogo

Processo:

1. De acordo com a posição da linha no arquivo de texto, e as informações de quantidade de pokémon, o programa lê essa linha e armazena, a partir da função strtok, nos atributos correspondentes.

2. O primeiro elemento da linha é adicionado a uma variável nome que mais tarde será copiado para o atributo nome do pokémon.

3. O segundo elemento é transformado em float pela função atof e atribuído ao poder de ataque do pokémon.

4. O terceiro elemento é transformado em float pela função atof e atribuído ao poder de defesa do pokémon.

5. O quarto elemento é transformado em float pela função atof e atribuído a vida inicial do pokémon.

6. O quinto e último elemento da linha é adicionado a uma variável tipo que mais tarde será copiado para o atributo tipo do pokémon.

7. Ao fim, o pokémon é armazenado na lista correspondente (dependendo dos valores do primeiro input). Internamente, após esse processo, o arquivo é fechado.

Output:
Jogador 1 venceu
Pokemon sobreviventes: Squirtle
Pokemon derrotados: Charmander

O caso exemplificado é o mais simples que pode acontecer, com cada um dos jogadores tendo apenas 1 pokémon para a batalha. Nesse caso, pode-se explicitar mais ainda o processamento:

1. Squirtle é atribuído como pokémon atacante e *atacanteDeQuem* como 0.

2. Charmander é atribuído como pokémon defensor *defensorDeQuem* como 1.

3. Como o tipo do atacante é água e o tipo do defensor é fogo, o ataque se torna mais forte e, pela função *ataque*, ele é aumentado em 20% (agora é 18)

4. A partir da função *vida*, como o valor de ataque é maior que o valor de defesa do defensor, é tirado da vida do defensor a quantidade de pontos referente a essa diferença (agora Charmander está com 30 de vida).

5. A partir das condicionais do programa, como Charmander não está com sua vida zerada, ele agora se torna o atacante e Squirtle o defensor, trocando também os valores de *atacanteDeQuem* e *defensorDeQuem*.

6. Pela função *ataque*, o ataque do turno diminui em 20% (20)

7. Pela função *vida*, como o ataque não é maior que a defesa, a vida de Squirtle diminui em 1 (40).

8. A partir das condicionais do programa, como Squirtle não está com sua vida zerada, ele agora se torna o atacante e Charmander o defensor , trocando também os valores de *atacanteDeQuem* e *defensorDeQuem*.

9. Pela função *ataque*, o ataque do turno aumenta em 20% (18)

10. Pela função *vida*, como o ataque é maior que a defesa, a vida de Charmander diminui de acordo com essa diferença (24).

11. Repetição dos passos de 5 a 7 (Vida de Squirtle: 39)

12. Repetição dos passos de 8 a 10 (Vida de Charmander: 12).

13. Repetição dos passos de 5 a 7 (Vida de Squirtle: 38).

14. Repetição dos passos de 8 a 10 (Vida de Charmander: 6).

15. Repetição dos passos de 5 a 7 (Vida de Squirtle: 37).

26. Repetição dos passos de 8 a 10 (Vida de Charmander: 0).

29. Como a vida do defensor agora é zero, mudamos o valor para 1 no seu índice em *derrotados* e diminuímos 1 na quantidade de pokémon do jogador correspondente (nesse

caso, o jogador 2). Caso o arquivo constasse mais pokémon, o próximo do jogador 2 se tornaria o atacante do próximo turno (sempre muda jogador atacante e jogador defensor).

30. Dessa vez, na conferência do while, o programa identifica que *quant2 == 0*, o que faz com que ele finalize o loop.

31. Como o *quant2 == 0*, é impresso na tela que o Jogador 1 venceu.

32. Por fim, imprime-se os pokémon não derrotados - sobreviventes - (Squirtle) e os derrotados (Charmander). Internamente, o espaço de memória alocado para as listas é liberado.

3. Testes e Erros

Durante o processo de elaboração do código, alguns erros foram encontrados, estabelecendo desafios de lógica de programação que exigiram contínuas mudanças no escopo do programa.

a. Processo de Criação do Algoritmo

A princípio, a organização do programa fez com que ele se tornasse confuso, portanto ao passo que o tempo passou, fui reorganizando-o em diferentes funções, dividindo as diferentes partes do código, a fim de torná-lo mais legível. Além disso, havia definido algumas das listas com tipos diferentes, de maneira que alguns erros gerais - não especificados - ocorriam e o programa não funcionava da maneira desejada. A partir do momento que defini as listas principais como do tipo Poke e a lista de controle de derrotados com valores booleanos, aos poucos esses problemas foram sendo resolvidos. Alguns erros mais marcantes durante o processo foram:

Erro 1: Alocação de memória

Correção: Estabelecimento de alocação dinâmica para as listas nomes e derrotados, de maneira que a alocação acontecia após a leitura da quantidade de pokémon que seriam armazenados.

Erro 2: Erro na impressão dos pokémon sobreviventes

Correção: Primeiramente, estava tentando criar uma lista para os sobreviventes e ir retirando ao passo que fossem derrotados; como não funcionou, modifiquei a lógica de programação dessa parte do programa para uma lógica mais simples, passando para uma lista de derrotados com valores booleanos, onde todos iniciavam com 0 e, ao passo que fossem derrotados, eram trocados por 1. Como o índice da lista derrotados era o mesmo da lista todos, ao fim bastava imprimir o nome da lista todos daqueles pokémonem que o índice constava como 0 na lista de derrotados (para os sobreviventes) ou 1 (para os derrotados).

Erro 3: Erro em alguns resultados de batalha (devido ao tipo)

Correção: O resultado encontrado na verdade foi bem simples. O que acontece é que, como o tipo é o último atributo dos pokémon, o caracter *\n* é adicionado ao final da string e, na hora da comparação para a manipulação do poder de ataque, isso ocasiona erros. Assim, a solução encontrada foi substituir o caracter citado por um *\0* ao fim da string, pela linha: *if (tipo[strlen(tipo) - 1] == '\n') tipo[strlen(tipo) - 1] = '\0'*.

Erro 4: Erro em batalhas de pokémon com nomes iguais Correção: Para a correção desse problema, foi necessário criar uma variável booleana - valores inteiros 1 ou 0 - em que 0 correspondia ao jogador 1 e 1 ao jogador 2 - como se perguntasse: pertence ao jogador 2? A partir da inicialização do processo de batalha, dependendo de a qual jogador pertence o pokémon defensor, essa variável se altera. Dessa forma, a conferência de quem é o pokémon defensor, que ocorria por nome do pokémon, passou a ser feita a partir dessa variável. Assim, o nome do pokémon defensor não importa mais, mas sim a quem ele pertence, de forma que as condicionais são mais eficazes para a realização das trocas de defensores e atacantes nos momentos oportunos.
Erro 5: E se a quantidade informada de pokémon não corresponder a quantidade apresentada no arquivo? Correção: Esse erro foi, mais que um erro, uma possibilidade de erro por parte da organização do usuário. Para resolvê-lo, assim que abro o arquivo conto a quantidade de linhas. Depois de saber quantos pokémon cada um dos jogadores deveriam ter (e, portanto, quantos deveriam ser apresentados no total), vejo se o último caracter do arquivo é um \n. Se for, quer dizer que a quantidade de linhas do arquivo menos 1 - que seria a linha que informa as quantidades - deve ser igual ao total de pokémon. Se não for, quer dizer que a contagem de linhas - que são contadas pelo \n - já é com 1 a menos. Assim, a quantidade de linhas deve corresponder ao total de pokémon. Se isso não acontecer, o programa retorna uma mensagem de erro ao usuário.

b. Testes Realizados

Para a percepção de erros e a correção desses, foram realizados vários testes, comparando os outputs do programa com os esperados. Segue um exemplo:

Input	**Output Esperado**	**Output do programa**
5 5 Squirtle 15 25 40 agua Sandshrew 17 13 28 pedra Charmeleon 23 15 24 fogo Raichu 19 16 28 eletrico Cloyster 59 12 51 gelo Charmander 25 12 36 fogo Pikachu 25 15 24 eletrico Dewgong 17 26 33 gelo Wartortle 26 41 19 agua Sandslash 62 31 40 pedra	Squirtle venceu Charmander Pikachu venceu Squirtle Sandshrew venceu Pikachu Dewgong venceu Sandshrew Charmeleon venceu Dewgong Wartortle venceu Charmeleon Wartortle venceu Raichu Cloyster venceu Wartortle Sandslash venceu Cloyster Jogador 2 venceu	Squirtle venceu Charmander Pikachu venceu Squirtle Pikachu venceu Sandshrew Charmeleon venceu Pikachu Charmeleon venceu Dewgong Wartortle venceu Charmeleon Wartortle venceu Raichu Cloyster venceu Wartortle Sandslash venceu Cloyster Jogador 2 venceu

A partir da percepção desse equívoco, foi possível procurar a fundo erros de lógica dentro do programa, o que garantiu que o código se tornasse mais eficiente e correto, permitindo as correções - no caso do exemplo, do erro 3 - citadas no tópico anterior. Para além, uma exemplificação de teste que garantiu que o programa estava 100% correto, já no final do processo de implementação, foi um teste que estabelecia para o programa um documento de texto com 26 pokémon para cada um dos jogadores, que envolvia nomes iguais de pokémon de diferentes jogadores. Com uma base maior de dados, foi possível garantir que todo o código estava correto e, a partir do momento que todas as saídas foram de acordo com as esperadas, confiar na implementação segura do programa.

4. Conclusão

O trabalho realizado destacou a importância da correta interpretação do problema e organização para o sucesso na programação. Entender plenamente os requisitos do projeto antes de iniciar a codificação é essencial. Além disso, a prática regular e a aprendizagem contínua são fundamentais para construir uma base sólida de conhecimentos, permitindo abordar problemas com confiança e eficiência. O aprimoramento constante é também crucial em um campo em constante evolução, e manter-se atualizado amplia o repertório de soluções. Vale também ressaltar que a persistência é chave para superar desafios, transformando erros em oportunidades valiosas de aprendizado. Ademais, a paciência é vital, pois momentos de dificuldade exige pausas para recarregar a criatividade e voltar ao trabalho com força renovada. Assim, com paciência, persistência e compromisso com o aprimoramento constante, é possível alcançar grandes realizações na programação.