

RELATÓRIO – CORREÇÃO DO ALGORITMO MERGESORT

Disciplina: Projeto e Análise de Algoritmos – 2025/2

Participantes:

- Ayrla Danielly Nascimento Costa - 190025069
- Geilson dos Santos Sá - 231006239
- Wesley Henrique Ferreira - 231021496

1 Introdução

Este projeto teve como principal objetivo aplicar os conceitos de lógica estudados em aula para formalizar e provar a correção de um algoritmo de ordenação utilizando o assistente de provas Coq. A proposta escolhida envolve a definição formal de ordenação de listas, a noção de permutação baseada em ocorrências e a prova de correção do algoritmo mergesort.

O relatório está dividido em três grandes etapas, seguindo a estrutura do projeto:

1. formalização da noção de lista ordenada e provas básicas sobre essa propriedade;
2. definição de permutação via contagem de ocorrências e demonstração de propriedades fundamentais;
3. formalização e prova de correção do algoritmo mergesort, mostrando que ele produz uma lista ordenada e que esta é uma permutação da lista de entrada.

Ao longo do trabalho, descrevemos as definições, dificuldades e estratégias utilizadas para construir as provas no Coq, desde os primeiros lemas elementares até as demonstrações mais complexas envolvendo funções recursivas sofisticadas como `merge` e `mergesort`.

2 Prova do Algoritmo

2.1 Primeira Etapa – Propriedades de Ordenação

A primeira etapa do projeto consistiu em compreender e trabalhar com a definição formal do predicado `sorted`, que descreve quando uma lista de números naturais está ordenada. Esse predicado possui três construtores: `nil_sorted`, `one_sorted` e `all_sorted`, e a maior parte da etapa inicial do trabalho envolve manipular essas três regras para provar lemas básicos.

Um dos primeiros desafios foi entender melhor como o construtor `all_sorted` funciona: para provar que $x :: y :: l$ está ordenada, é necessário demonstrar que $x \leq y$ e que $y :: l$ é ordenada. A partir disso, vários lemas foram desenvolvidos, como:

- **tail_sorted**: remover o primeiro elemento de uma lista ordenada mantém a ordenação;
- lemas sobre o predicado `le_all`, que formaliza “um elemento é menor ou igual a todos de uma lista”;
- propriedades envolvendo casos específicos da estrutura de listas ordenadas.

Primeiramente exigiu-se provar que, se um elemento a é menor ou igual a todos os elementos de l e l é ordenada, então $a :: l$ também é ordenada. A prova foi construída usando indução e aplicação direta da regra `all_sorted`.

Em seguida invertemos o raciocínio: Agora queremos mostrar que, se $a :: l$ é ordenada, então $a \leq^* l$. Essa demonstração também foi feita por indução na lista, exigindo uma boa compreensão de como `sorted` se decompõe em seus casos.

Esses lemas iniciais foram essenciais, pois estabeleceram a base lógica necessária para avançar para o algoritmo `merge`.

2.2 Segunda Etapa – Permutação e Contagem de Ocorrências

A segunda etapa introduziu a noção de permutação, mas diferente de outras abordagens, aqui a permutação é definida a partir da função `num_oc`, que conta o número de ocorrências de um elemento em uma lista.

Duas listas são permutações uma da outra se todo elemento natural possui exatamente o mesmo número de ocorrências em ambas. Essa definição permitiu desenvolver propriedades fundamentais, como:

- reflexividade da permutação;

- comportamento de `num_oc` em `append`, mostrando que a contagem em `l1 ++ l2` é a soma das contagens em cada lista;
- lemas técnicos que prepararam terreno para analisar `merge` e `mergesort`.

Essa etapa é crucial, pois a correção de `mergesort` depende não apenas de provar ordenação, mas também de que a função não perde nem duplica elementos da entrada.

2.3 Terceira Etapa – Definição e Propriedades do `merge` e `mergesort`

A terceira etapa foi a mais trabalhosa, pois envolve lidar com funções recursivas mais sofisticadas, definidas com a palavra-chave `Function`, que requer a prova de que a métrica diminui a cada chamada recursiva.

Função `merge` A função `merge` recebe duas listas ordenadas e produz uma nova lista ordenada contendo todos os elementos das duas. A métrica utilizada foi a soma dos tamanhos das listas, o que garante que cada chamada recursiva reduz o “tamanho total” do par.

Um lema central deste trecho foi `merge_in`, que mostra que qualquer elemento da lista resultante de `merge` pertence a uma das listas originais, essencial para provar a permutação mais adiante.

2.3.1 – Provar que `merge` preserva ordenação Essa foi, sem dúvida, uma das partes mais complexas. A prova envolve quatro casos correspondentes às diferentes formas que o par de listas pode assumir:

- primeira lista vazia;
- segunda lista vazia;
- cabeças iguais ou uma delas menor;
- casos recursivos onde é necessário aplicar hipóteses de indução.

Dois desses casos foram fornecidos como exemplo, e os outros dois precisaram ser provados seguindo a mesma estratégia. Levaram bastante tempo por exigirem reconstruir manualmente a estrutura da chamada recursiva.

Mergesort A função `mergesort` também exige uma métrica, já que divide a lista, chama-se recursivamente e depois utiliza `merge` para recombinar.

2.3.2 – `mergesort` realmente ordena a lista A prova usa:

- a hipótese de indução para cada sublista;

- o fato de que `merge` ordena quando recebe listas ordenadas.

2.3.3 – mergesort preserva número de ocorrências Essa etapa depende fortemente dos lemas sobre `num_oc` e, novamente, do lema `merge_in`. Mostra que nenhum elemento é perdido ou acrescentado ao longo do processo, com base na definição de permutação.

2.3.4 – Correção de mergesort Com as duas partes estabelecidas, ordenação e permutação, conclui-se formalmente que `mergesort` está correto.

2.4 Experiência no Desenvolvimento

O desenvolvimento do projeto exigiu traduzir raciocínios matemáticos aparentemente simples em provas detalhadas no Coq. Muitas propriedades intuitivas precisaram ser decompostas em lemas menores, e pequenos detalhes da sintaxe e do sistema exigiram atenção.

O processo foi importante para compreender:

- como estruturar provas formais;
- como construir definições robustas;
- como dividir um problema complexo em partes menores.

O trabalho com funções recursivas definidas por `Function` adicionou uma dificuldade extra, pois cada chamada recursiva precisava ser justificada pelo decrescimento da métrica. Isso tornou as provas de `merge` e `mergesort` particularmente desafiadoras.

Durante o desenvolvimento do projeto, como sugerido pelo professor, utilizamos ferramentas de apoio baseadas em IA (como o ChatGPT) para auxiliar na formulação de ideias de prova e esclarecimento conceitual, especialmente no teorema:

```
merge_sorts: forall p, sorted_pair_lst p -> sorted (merge p)
```

que foi uma etapa complexa da atividade.

3 Conclusão

O principal aprendizado do projeto foi perceber como a divisão em lemas auxiliares é fundamental para provar propriedades mais complexas. A formalização da noção de lista ordenada, da relação de permutação e das funções recursivas mostrou que mesmo algoritmos clássicos, como `mergesort`, exigem argumentação rigorosa para serem totalmente verificados.

A combinação entre:

- provas de ordenação,
- verificação de preservação dos elementos,
- e raciocínio recursivo fundamentado,

permitiu concluir, de forma formal e precisa, que o algoritmo `mergesort` é correto segundo a definição adotada.

Além de reforçar conceitos de lógica e prova formal, o projeto demonstrou a importância de detalhar cada passo e a utilidade prática de decompor problemas grandes em partes menores e mais manejáveis.