PROJETO E ANÁLISE DE ALGORITMOS

Alisson Vieira Neves - 221002067
Lucas da Costa Rodrigues - 221017079

1. Introdução

O projeto visa provar a correção do algoritmo Binary Insert Sort, cuja estrutura básica inicial foi fornecida pelo professor através de seu github, e não foi alterada. Para provar a correção do algoritmo, buscou-se provar, primeiro, a correção de suas principais funcionalidades, dispondo de todo o conhecimento adquirido no decorrer do semestre. O maior desafio encontrado para tal, foi a familiarização com a ferramenta de prova, para o qual se contou com a ajuda de modelos de inteligência artificial.

2. Funcionamento de Binary Insertion Sort

Este algoritmo de ordenação funciona de maneira muito similar ao algoritmo de ordenação Insertion Sort, sua principal diferença sendo, assim como evidenciado pelo seu nome, a utilização do algoritmo de busca binária, ao invés do algoritmo de busca linear, para encontrar a posição correta de determinado item. Assim sendo, para que sua correção fosse devidamente provada, foi decidido que seria necessário provar que a inserção de um elemento preserva a estrutura da lista.

3. Implementação

A prova do algoritmo binsertion_sort se dá através de 3 algoritmos, primeiramente usamos insert_at_perm, logo depois usamos binsert_perm e para finalizar utilizamos binsertion_sort_correct. Uma análise detalhada de cada um deles pode ser conferida abaixo.

3.1. Insert_at_perm:

Deseja-se provar que inserir um elemento x em qualquer posição i de uma lista l resulta apenas numa permutação (rearranjo) da lista original com o x adicionado. Como a função insert_at é recursiva baseada no índice i (ela conta regressivamente até zero), usamos a Indução no índice i.

3.1.1. Caso Base:

O índice é 0 (i = 0): neste caso a função insert_at 0 x l simplesmente insere x na cabeça da lista (x :: l). Para provar que a lista x :: l é uma permutação de x :: l,basta perceber que a propriedade da Reflexividade nos garante isto, já que qualquer coisa é permutação dela mesma.

3.1.2. Passo Indutivo:

O índice é positivo (i = k + 1): Assumimos como verdade que inserir x na posição k (uma posição anterior) gera uma permutação válida em qualquer lista. Para avançar, precisamos olhar para a lista l. Dividimos em dois sub-casos, descritos abaixo.

3.1.2.1. A lista é vazia

A função insert_at tem uma proteção: se o índice for maior que 0 mas a lista for vazia, ela retorna [x]. Trivialmente, pela propriedade da reflexividade, [x] é permutação de [x].

3.1.2.2. A lista tem elementos

Neste caso, existem duas possibilidades: ou o alvo foi encontrado e x é inserido na cabeça da lista [x; h; ...resto], ou a função precisa ser chamada recursivamente [h; (x inserido no resto)...]. Para provar que [h; (x inserido no resto)...] é permutação de [h; (x inserido no resto)...], foi utilizada uma estratégia de "Ponte" (Transitividade) em duas etapas:

3.1.2.2.1. **Troca (Swap)**

Pegamos o Lado Esquerdo [x; h; ...] e trocamos os dois primeiros elementos de lugar. Isso vira [h; x; ...]. A teoria das permutações garante que trocar dois vizinhos é válido (perm_swap).

3.1.2.2.2. **Salto (Skip)** e a Hipótese

Agora comparamos o resultado da troca com o Lado Direito, temos [h; x; ... tl], e queremos [h; (insert_at k x tl)]. Como ambos começam com o elemento h, podemos ignorar a cabeça (perm_skip) e focar apenas nas caudas. Agora precisamos provar que [x; ...tl] é permutação de (insert_at k x tl). Como assumimos que funcionava para o índice k, a prova está concluída.

3.2. Binsert_perm:

O funcionamento da função binsert pode ser decomposto em dois passos, primeiro ela calcula uma posição usando bsearch (p), depois chama a função insert_at na posição p. Assim, o problema se transforma simplesmente em provar que *x::l* é uma permutação de *insert_at p x l*.O lema insert_at_perm, que já fora anteriormente

provado, diz que "Para qualquer número i, inserir x na posição i gera uma permutação válida." Como este lema vale para todo e qualquer número, ele automaticamente vale para o número específico (p), que bsearch calculou.

3.3. Binsertion_sort_correct

Queremos provar que o algoritmo binsertion_sort satisfaz duas condições para qualquer lista l, primeiro, queremos saber se binsert a (binsertion_sort l') resulta numa lista ordenada. O teorema auxiliar binsert_correct (que provamos anteriormente) diz: *"Se você inserir um elemento numa lista que já é ordenada, o resultado continua ordenado"*. A lista onde estamos inserindo é binsertion_sort l'. Sabemos que tal lista é ordenada, pois a nossa Hipótese de Indução garante isso, logo, o resultado final também é ordenado.

Agora queremos provar que a lista original a :: l' é uma permutação do resultado final binsert a (binsertion_sort l'). Para conectar o início ao fim, usamos uma estratégia de ponte (Transitividade) passando por um estado intermediário:

1. Estado Inicial: a :: l'
2. Estado Intermediário: a :: (binsertion_sort l') (A cabeça original, colada na cauda já ordenada).
3. Estado Final: binsert a (binsertion_sort l') (A cabeça inserida no lugar certo).

Passo 1: Do Inicial para o Intermediário

- Comparamos a :: l' com a :: (binsertion_sort l').
- Como a cabeça a é igual em ambos, podemos ignorá-la (perm_skip) e olhar só para as caudas.
- As caudas são permutações uma da outra? Sim, a Hipótese de Indução garante que l' é permutação de binsertion_sort l'.

Passo 2: Do Intermediário para o Final

- Comparamos a :: (binsertion_sort l') com binsert a (binsertion_sort l').
- Isso é exatamente o que o lema auxiliar binsert_perm prova: *"Colocar um elemento na cabeça ou inseri-lo na posição ordenada resulta na mesma coleção de elementos"*.
- Logo, a permutação se mantém.

4. Desenvolvimento do projeto

Para o desenvolvimento do projeto utilizamos o assistente de prova Coq, a instalação foi o ponto de maior complicação na utilização do assistente de provas. Para provar o teorema binsertion_sort_correct utilizamos o código fornecido pelo professor e utilizamos uma inteligência artificial generativa para ajudar no entendimento dos comandos utilizados pelo Coq, para o projeto decidimos fazer um arquivo separado com a prova principal e as provas auxiliares e alguns testes para ver se o algoritmo estava funcionando.

Como descrito no projeto, dividimos a prova nos algoritmos insert_at_perm, binsert_perm e binsertion_sort_correct. A funcionalidade de cada algoritmo já foi descrita, mas em resumo, o primeiro prova que a operação física de inserir um elemento em uma lista nunca cria nem destrói dados, o segundo prova que a busca binária não afeta a integridade dos dados, o terceiro prova que ao aplicar binsert repetidamente em uma lista , o resultado final é garantidamente ordenado e contém os mesmos elementos da entrada.

Ao notarmos que a função insert_at diminui o índice i passo a passo, a melhor decisão foi usar indução no i, o que resolve trivialmente o caso onde i=0, pois inserir na posição 0 é apenas colocar o elemento na frente. No passo recursivo a nossa lista começa com o elemento novo x (x :: h :: tl), mas a função recursiva manteve o elemento original h na frente (h :: ...). Para resolver isso usamos a transitividade para fazer uma troca: invertemos o x com o h na lista da esquerda, agora as listas começam com h. Visto isso, podemos usar a regra perm_skip para cancelar essas cabeças. O que resta nas caudas é exatamente a definição da nossa hipótese de indução.

O próximo passo foi ver como a função binsert é feita, assim foi possível perceber que ela é composta por duas etapas: primeiro um cálculo de posição (bsearch), e depois a execução da inserção (insert_at) naquela posição. Não é preciso entender a lógica da busca binária. O cálculo do bsearch resulta apenas em um número natural (um índice). Como o nosso lema anterior (insert_at_perm) provou que a inserção preserva os elementos para qualquer índice i, ele se torna uma regra universal. Se a inserção é uma permutação válida para todo número, então ela também é válida para o número específico calculado pelo bsearch. Assim ao aplicar o lema anterior o Coq aceita a prova sem olhar para a complexidade da busca.

Como o algoritmo ordena a lista processando um elemento de cada vez, o melhor seria seguir pela indução. O caso base é aceito imediatamente, pois é uma lista vazia. O foco principal é o passo indutivo, assumimos que a cauda da lista já foi ordenada corretamente pela recursão e precisamos provar que adicionar a cabeça mantém tudo certo. A nossa hipotese de indução já garante que a cauda está ordenada, o binsert_correct garante que inserir um elemento em uma lista ordenada resulta em uma nova lista ordenada, assim só é preciso ligar essas duas afirmações para garantir que a lista final está ordenada.

Para provar que os dados não foram corrompidos, construímos uma ponte que liga início e o fim usando transitividade, para isso comparamos a lista original (a :: l') com uma versão onde a cauda já foi ordenada (a :: binsertion_sort l'). Como a cabeça é imóvel, focamos

apenas na cauda, e a hipótese de indução nos garante que a cauda ordenada é uma permutação válida da original. Após isso é preciso mover o item(a) da cabeça para o meio da lista ordenada assim invocamos o lema binsert_perm, que garante que essa inserção é apenas um rearranjo seguro. Ao juntar essas duas partes, provamos que o algoritmo não apenas ordena, mas preserva a integridade dos dados originais.