# Lógica Computacional 1
# Descrição do Projeto

Flávio L. C. de Moura

10 de junho de 2025

## 1 Introdução

O objetivo deste projeto é utilizarmos a(s) lógica(s) estudadas durante o curso para resolvermos alguns problemas não triviais em Computação, como por exemplo, provar a correção de um algoritmo de ordenação. A presente proposta contém duas sugestões de trabalho, e cada dupla deve escolher (pelo menos) uma:

1. Estabelecer a equivalência entre três definições distintas de ordenação de listas de números naturais;
2. Provar a correção do algoritmo de ordenação por insersão.

## 2 Proposta1: Equivalência entre diferentes noções de ordenação.

A primeira definição de ordenação, chamada *ord1*, é uma definição indutiva contendo 3 regras de formação:

$$\frac{}{ord1\ nil}\ (ord1\_nil) \qquad \frac{}{ord1\ (x :: nil)}\ (ord1\_one) \qquad \frac{x \leq y \qquad ord1(y :: l)}{ord1\ (x :: y :: l)}\ (ord1\_all)$$

```
Inductive ord1 : list nat → Prop :=
| ord1_nil: ord1 nil
| ord1_one: ∀ x, ord1 (x::nil)
| ord1_all: ∀ l x y, x ≤ y → ord1 (y::l) → ord1 (x::y::l).
```

A segunda definição de ordenação, chamada *ord2*, é uma definição indutiva contendo 2 regras de formação:

$$\frac{}{ord2\ nil}\ (ord2\_nil) \qquad \frac{x \leq^* l \qquad ord2\ l}{ord2\ (x :: l)}\ (ord2\_all)$$

onde $x \leq^* l$ significa que $x$ é menor ou igual que todo elemento da lista $l$. Formalmente, este predicado é definido como a seguir:

```
Definition le_all x l := ∀ y, In y l → x ≤ y.
Inductive ord2 : list nat → Prop :=
| ord2_nil: ord2 nil
| ord2_all: ∀ l x, x ≤* l → ord2 l → ord2 (x::l).
```

A terceira definição de ordenação, chamada *ord3*, diz que uma lista está ordenada se cada elemento é menor ou igual ao elemento seguinte:

```
Definition ord3 (l : list nat) : Prop := ∀ i, i < length l → nth i l 0 ≤ nth (S i) l 0.
```

A quarta definição de ordenação, chamada *ord4*, diz que uma lista está ordenada se cada elemento é menor ou igual que qualquer elemento que esteja em posição anterior:

```
Definition ord4 (l : list nat) : Prop := ∀ i j, i < j → j < length l → nth i l 0 ≤ nth j l 0.
Lemma ord3_nil: ord3 nil.
```

Os 4 lemas a seguir são exercícios simples que serão provados em aula:

```
Lemma ord3_one: ∀ x, ord3 (x::nil).
Lemma ord4_nil: ord4 nil.
Lemma ord4_one: ∀ x, ord4 (x::nil).
```

**O objetivo da proposta 1 é mostrar que as definições *ord1*, *ord2*, *ord3* e *ord4* são equivalentes. Para isto prove os 6 teoremas a seguir**:

```
Theorem ord1_equiv_ord2: ∀ l, ord1 l ↔ ord2 l.
Theorem ord1_equiv_ord3: ∀ l, ord1 l ↔ ord3 l.
Theorem ord1_equiv_ord4: ∀ l, ord1 l ↔ ord4 l.
Theorem ord2_equiv_ord3: ∀ l, ord2 l ↔ ord3 l.
Theorem ord2_equiv_ord4: ∀ l, ord2 l ↔ ord4 l.
Theorem ord3_equiv_ord4: ∀ l, ord3 l ↔ ord4 l.
```

# 3 Proposta 2: O algoritmo de ordenação por inserção

O algoritmo de ordenação por inserção é baseado em na função auxiliar *insert* que definimos a seguir. A função *insert* recebe um natural $x$ e uma lista $l$ como argumentos, e é definida recursivamente na estrutura de $l$:

```
Fixpoint insert (x:nat) (l: list nat) :=
  match l with
  | nil ⇒ x :: nil
  | h :: tl ⇒ if x ≤? h then (x :: l)
                else (h :: (insert x tl))
  end.
```

Como podemos observar, quando a lista $l$ é a lista vazia, a função retorna a lista unitária contendo o elemento que foi inserido. Quando $l$ não é a lista vazia, então ela tem a forma $h$::$tl$, isto é, $l$ tem $h$ como primeiro elemento, e $tl$ como cauda. Para saber onde inserir um elemento $x$, comparamos $x$ com $h$, e quando $x$ é menor ou igual a $h$ simplesmente inserimos $x$ na primeira

posição da lista. Caso contrário, a função vai recursivamente encontrar a posição correta para inserir $x$.

A função principal do algoritmo é dada a seguir:

```
Fixpoint insertion_sort l :=
  match l with
  | nil ⇒l
  | h :: tl ⇒ insert h (insertion_sort tl)
  end.
```

A função *insertion_sort* é definida recursivamente na estrutura da lista $l$ que é dada como argumento. Quando a lista é vazia não há nada a fazer, e caso contrário, a função *insert* é chamada para inserir a cabeça $h$ da lista na cauda $tl$ onde a função é aplicada recursivamente.

O primeiro passo para estebelecermos a correção do algoritmo de ordenação por inserção consiste em mostrar que a função *insertion_sort* retorna uma lista ordenada:

```
Lemma insertion_sort_sorts: ∀ l, ord1 (insertion_sort l).
```

Observe que no lema anterior, qualquer das noções de ordenação utilizadas no arquivo *ord_equiv* podem ser utilizadas.

O segundo passo consiste em mostrar que o algoritmo gera uma permutação da lista de entrada. No arquivo *perm_equiv* apresentamos duas definições distintas de permutação e provamos que são equivalentes. Utilize qualquer uma destas definições e mostre que a saída do algoritmo de ordenação por inserção é uma permutação da lista de entrada:

```
Lemma insertion_sort_permutes: ∀ l, Permutation l (insertion_sort l).
```

Por fim, a correção do algoritmo de ordenção por inserção é dada pelo seguinte teorema *insertion_sort_correct* abaixo. **O objetivo da proposta 2 é provar o teorema** *insertion_sort_correct*:

```
Theorem insertion_sort_correct: ∀ l, ord1 (insertion_sort l) ∧ Permutation l (insertion_sort l).
```

onde o predicado *Permutation* é definido pelas regras a seguir:

$$\frac{}{Permutation\ nil\ nil}\ (perm\_nil) \qquad \frac{Permutation\ l\ l'}{Permutation\ (x :: l)\ (x :: l')}\ (perm\_skip)$$

$$\frac{}{Permutation\ (y :: x :: l)\ (x :: y :: l)}\ (perm\_swap)$$

$$\frac{Permutation\ l\ l' \qquad Permutation\ l'\ l''}{Permutation\ l\ l''}\ (perm\_trans)$$

onde $x$, $y$, $l$, $l'$ e $l''$ são variáveis universais.

O código Coq correspondente a essas regras é listado a seguir:

```
Inductive Permutation (A : Type) : list A -> list A -> Prop :=
 | perm_nil : Permutation nil nil
 | perm_skip : forall (x : A) (l l' : list A),
               Permutation l l' ->
```

```
                Permutation (x :: l) (x :: l')
  | perm_swap : forall (x y : A) (l : list A),
                Permutation (y :: x :: l)
                  (x :: y :: l)
  | perm_trans : forall l l' l'' : list A,
                 Permutation l l' ->
                 Permutation l' l'' ->
                 Permutation l l''.
```

Alternativamente, a lista $l$ é uma permutação da lista $l'$ se ambas possuem os mesmos elementos. A implementação dessa ideia é feita baseada no número de ocorrência de cada elemento na lista. Assim, na definição a seguir, $num\_oc\ x\ l$ retorna o número de ocorrências do elemento $x$ na lista $l$:

```
Fixpoint num_oc x l :=
  match l with
    | nil ⇒ 0
    | h :: tl ⇒
      if x =? h then S(num_oc x tl) else num_oc x tl
  end.
```

A definição *equiv* a seguir, expressa que a lista $l$ é uma permutação da lista $l'$ se ambas possuem o mesmo número de ocorrências de qualquer elemento:

```
Definition equiv l l' := ∀ n:nat, num_oc n l = num_oc n l'.
```

O teorema a seguir formaliza que as definições *Permutation* e *equiv* são equivalentes, e portanto qualquer uma delas pode ser utilizada no projeto.

```
Theorem perm_equiv: ∀ l l', Permutation l l' ↔ equiv l l'.
```

# 4 Etapas de desenvolvimento do Projeto

O projeto pode ser feito individualmente ou em dupla. As duplas serão organizadas no GitHub Classroom da seguinte forma:

- Depois de formadas as duplas, um dos membros do grupo vai acessar o link

  classroom.github.com/a/rBuwfZii

  e criar o grupo (clicando no botão "Create team");

- Depois que o grupo foi criado, o outro participante precisa acessar o mesmo link e clicar no botão "Join" do grupo correspondente.

**Importante!** Siga rigorosamente os passos acima para a formação dos grupo, e observe que depois que os grupos foram formados, não é possível fazer a migração para outros grupos.

O projeto será dividido em duas etapas como segue:

- **Verificação da Formalização** (20 pontos): Os grupos deverão completar todas as provas dos arquivos `ord_equiv.v` e/ou `ins_sort.v`.

- **Entrega do Relatório Final** (10 pontos): Cada grupo de trabalho deverá entregar um relatório inédito (em formato pdf), limitado a oito páginas (12 pts, A4, espaçamento simples). No relatório o grupo deverá traduzir as provas feitas para linguagem natural, e relatar a experiência no desenvolvimento do projeto: dificuldades, aprendizados, sugestões, etc. Uma cópia deste relatório deve ficar no repositório do projeto no GitHub.

- O prazo para finalização do projeto é 20/julho/2025. Nesta data, os arquivos `ord_equiv.v` e/ou `ins_sort.v`, assim como o relatório final em formato PDF, devem estar atualizados no repositório!

# Referências


[ARdM17] M. Ayala-Rincón and F.L.C. de Moura. *Applied Logic for Computer Scientists - computational deduction and formal proofs*. UTiCS, Springer, 2017.

[BvG99] S. Baase and A. van Gelder. *Computer Algorithms — Introduction to Design and Analysis*. Addison-Wesley, 1999.

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series. MIT press, third edition, 2009.