# Advanced Compiler 2024 - HW1 Data Dependence Analysis

## Solving diophantine equation to find dependency

I find dependencies by solving the diophantine equation. The following is my process:

1. Use *diophantine_solver()* to get the result *struct SolEquStru f*.
2. Determine the upper and lower bounds of the x-factor and y-factor, then find the intersection of the two ranges.
3. Finally, checking for dependencies within the interval.

## Bonus

### Mixin pattern (5%)

LLVM uses the mixin pattern with CRTP (Curiously Recurring Template Pattern) for its pass classes to allow flexibility and reusability in its design. This approach makes it easier to extend or customize passes, as new functionality can be mixed in as needed. The mixin pattern also enables more modular code, allowing different passes to share behavior in a way that is both efficient and maintainable.

### Utilize ADT (5%)

I use llvm::SmallVector to replace the std::vector in my code. It can allocate space for some number of elements(N), so if the actual number doesn't exceed the number N, it doesn't perform the dynamic allocation; that is, if the number of elements is "usually small", it would be a better choice than std::vector. In assignment 1, the number of statements is small, so I used the llvm::SmallVector.