- Solution

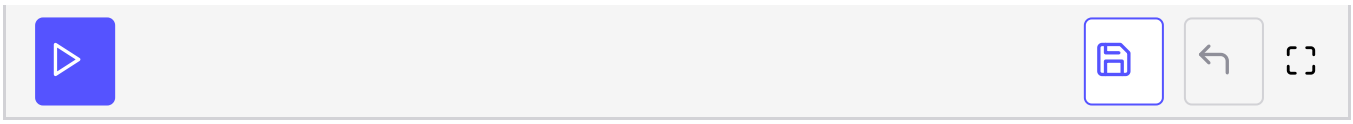
In this lesson, we'll look at the solution review of the last exercise.

WE'LL COVER THE FOLLOWING

Solution Review

```
// gcdVariation.cpp

#include <iostream>
#include <type_traits>
#include <typeinfo>

template<typename T1, typename T2,
         typename R = typename std::conditional <(sizeof(T1) < sizeof(T2)), T1, T2>::type>
R gcdConditional(T1 a, T2 b){
  static_assert(std::is_integral<T1>::value, "T1 should be an integral type!");
  static_assert(std::is_integral<T2>::value, "T2 should be an integral type!");
  if( b == 0 ){ return a; }
  else{
    return gcdConditional(b, a % b);
  }
}


template<typename T1, typename T2,
         typename R = typename std::common_type<T1, T2>::type>
R gcdCommon(T1 a, T2 b){
  static_assert(std::is_integral<T1>::value, "T1 should be an integral type!");
  static_assert(std::is_integral<T2>::value, "T2 should be an integral type!");
  if( b == 0 ){ return a; }
  else{
    return gcdCommon(b, a % b);
  }
}

int main(){


  std::cout << std::endl;

  std::cout << "gcdConditional(100, 10LL) = " << gcdConditional(100, 10LL) << std::endl;
```

```
  std::cout << "gcdCommon(100, 10LL) = " << gcdCommon(100, 10LL) << std::endl;

  std::conditional <(sizeof(int) < sizeof(long long)), int, long long>::type gcd1 = gcdCondit

  auto gcd2 = gcdCommon(100, 10LL);

  std::cout << std::endl;

  std::cout << "typeid(gcd1).name() = " << typeid(gcd1).name() << std::endl;
  std::cout << "typeid(gcd2).name() = " << typeid(gcd2).name() << std::endl;

  std::cout << std::endl;

}
```



Explanation

In the above code, we have defined an automatic return type `R` which returns data based on the data type passed in the function `gcdCommon` and `gcdConditional`.

Automatic return type deduction can't be used in both algorithms, because both algorithms solve their job by recursion (lines 24 and 26). The critical observation is that this recursion swaps their arguments. For example, `gcdCommon(a, b)` invokes `gcdCommon(b, a % b)`. This recursion, therefore, creates a function with different return types. Having a function with different return types is not valid.

Let's move on to template metaprogramming in the next lesson.