Wire-compatible Protocol buffer

Input file:	`standard input`
Output file:	`standard output`
Time limit:	3 seconds
Memory limit:	256 megabytes

Protocol buffers (or protobuf) are a flexible, efficient, automated mechanism for serializing structured data – think XML, but smaller, faster and simpler. In this problem, we will discuss the wile-compatible problem on a simplified protocol buffer.

Protobuf descriptor

Unlike XML or JSON, which is schema-less, protobuf has a schema. Before actually serializing data, you need to define the structure of the data you'd like to serialize in the descriptor file. On the left side of the following is a basic example of a protobuf descriptor that defines a message containing information about a Car.

```
message Car {
    required string brand = 1 ;
    required string model = 2 ;
    required double max_mileage = 4 ;
    optional double max_speed = 3 ;

    repeated Accessory accessories = 7 ;
}

message Accessory {
    required string name = 1 ;
    optional string description = 2 ;
    required double price = 3 ;
}
```

```
brand: "Tesla"
model: "Model 3"
max_mileage: 595
max_speed: 261
accessories {
    name: "Body painting"
    desctiption: "(Blue)"
    price: 10000.0
}
accessories {
    name: "Autopilot model"
    price: 56000.0
}
```

Given the protobuf descriptor, we can define protobuf instances, for example, an instance of a Car protobuf is on the right side of the above.

As you can see, the message format is simple – each message type has one or more uniquely numbered fields, and each field has a name and a value type, where value types can be integers float numbers, booleans, strings, raw bytes, or even other protocol buffer message types (to simplify, in this problem we only consider doubles, strings and embedded protobuf messages like `Accessory` in the example), allowing you to structure your data hierarchically.

The descriptor of protobuf contains a set of messages. Each message contains a set of numbered fields. Each field has its own field rules (required, optional or repeated), type, and a field name (e.g. `brand`, `model`, `max_mileage` in the example).

Field rules

Each field has one of the following rules:

- required: A well-formed message must have exactly one of this field.
- optional: A well-formed message can have zero or one of this field (but no more than one).
- repeated: This field can be repeated any number of times (including zero) in a well-formed message. The order of the repeated values will be preserved.

Field types

In this problem, we only consider the following types:

- double: 64 bit IEEE 754 double-precision Float Point Number.
- string: UTF-8 encoded string of any length.
- message: other protobuf message types defined in the descriptor.

Field numbers

Each field is assigned with a unique number, which is an integer between 1 and $2^{29} - 1$, or 536,870,911.

Syntax

Formally, the syntax of the simplified protobuf can be specified using Extended Backus-Naur Form (EBNF):

```
|   alternation
()  grouping
[]  option (zero or one time)
{}  repetition (any number of times)

letter = "A" ... "Z" | "a" ... "z"
digit = "0" ... "9"

ident = letter { letter | digit | "_" }
messageName = ident
fieldName = ident
messageType = messageName

decimal = ("1" ... "9") { digit }
fieldNumber = decimal

label = "required" | "optional" | "repeated"
type = "double" | "string" | messageType
field = label type fieldName "=" fieldNumber ";"
messageBody = "{" { field } "}"
message = "message" messageName messageBody

descriptor = { message }
```

Note: message names and field names are case sensitive and cannot be reserved words ("message", "required", "optional", "repeated", "double", "string").

Protobuf encoding

In this section, we will discuss how to serialize protobuf instances into bytes.

Here is a simple example. Given a simple protobuf descriptor:

```
message Test1 {
    optional double a = 1 ;
    repeated string b = 2 ;
}
```

And an instance:

```
a: 1.0
b: "abc"
b: "ABC"
```

Its wire-format encoding is:

```
09 3f f0 00 00 00 00 00 00 12 03 61 62 63 12 03 41 42 43
```

Base 128 Varints

In this section, we will take a look at how an instance is encoded into a byte stream.

To understand a simple protocol buffer encoding, you first need to understand varints. Varints are a method of serializing integers using one or more bytes. Smaller numbers take a smaller number of bytes.

Each byte in a varint, except the last byte, has the most significant bit (msb) set – this indicates there are further bytes to come. The lower 7 bits of each byte are used to store the two's complement representation of the number in groups of 7 bits, least significant group first.

So, for example, here is the number 1 – it's a single byte, so the msb isn't set:

```
0000 0001
```

And here is 300 – this is a bit more complicated:

```
1010 1100 0000 0010
```

How do you figure out this is 300? First, you drop the msb from each byte, as this is just there to tell us whether we've reached the end of the number (as you can see, it's set in the first byte as there is more than one byte in the varint):

```
    1010 1100 0000 0010
->   010 1100  000 0010
```

You reverse the two groups of 7 bits because, as you remember, varints store numbers with the least significant group first. Then you concatenate them to get your final value:

```
000 0010  010 1100
->  000 0010 ++ 010 1100
->  100101100
->  256 + 32 + 8 + 4 = 300
```

Message structure

As you know, a protocol buffer message is a series of key-value pairs. The binary version of a message just uses the field's number as the key – the name and declared type for each field can only be determined on the decoding end by referencing the message type's descriptor.

When a message is encoded, the keys and values are concatenated into a byte stream. The "key" for each pair in a wire-format message is actually two values – the field number from your protobuf descriptor file, plus a wire type that provides just enough information to find the length of the following value. In most language implementations this key is referred to as a tag.

The available wire types are as follows:

Type	Meaning	Used For
0	Varint	int32, etc. (Not related to this problem)
1	64-bit	double, etc.
2	Length-delimited	strings, embedded messages, etc.
3	Start group	groups. (Not related to this problem)
4	End group	groups. (Not related to this problem)
5	32-bit	float, etc. (Not related to this problem)

Each key in the streamed message is a varint with the value

```
(field_number << 3) | wire_type
```

Now let's look at our simple example again. You now know the first number in the stream is always a varint key, and here it's 09, or (dropping the msb):

```
000 1001
```

You take the last three bits to get the wire type (1) and then right-shift by three to get the field number (1). So you now know the field number is 1 and the following value is a 64-bit. Then, according to the descriptor, we know the following 8 bytes:

```
3f f0 00 00 00 00 00 00
```

is a double which is the representation of 1.0 in IEEE 754.

String

A wire type of 2 (length-delimited) means the value is a varint encoded length followed by the specified number of bytes of data.

Let's look at simple examples again:

```
12 03 61 62 63
12 03 41 42 43
```

The key here is 0x12, where field number = 2, wire type = 2. The length varint is 3, and then 0x61 0x62 0x63 is the ascii code for "abc", and 0x41 0x42 0x43 is the ascii code for "ABC".

Embedded messages

Here's a message descriptor with an embedded message of our example type, Test2:

```
message Test2 {
  optional Test1 c = 1;
}
```

And here's the encoded version, with Test1's a field set to 1.0:

```
0a 09 09 3f f0 00 00 00 00 00 00
```

The key is 0x0a, field number = 1, wire type = 2. The length varint is 9, and then the following 9 bytes is a Test1 instance, with a = 1.0.

Optional and repeated elements

If the protobuf descriptor has repeated elements, the encoded message can have zero or more key-value pairs with the same field number. These repeated values do not have to appear consecutively; they may be interleaved with other fields. The order of the elements with respect to each other is preserved when parsing.

For any optional fields, the encoded message may or may not have a key-value pair with that field number.

Field Order

Field numbers may be used in any order in a descriptor. The order chosen has no effect on how the messages are serialized.

Wire-format compatible problem

If message A is wire-format compatible with message B, it means the serialized bytes of any instance of message A can be parsed by message B without breaking the protobuf encoding assumption (and no unknown fields), and vice versa.

Input

The first line contains an integer n indicates the number of lines of a protobuf descriptor.

The following n lines are the content of the protobuf descriptor which contains a set of messages. Each line will not contain more than 120 characters.

Then follows a line with an integer m ($1 \leq m \leq 50000$), indicating there are m wire-format compatibility queries.

Then follow m lines, each of which is a wire-format compatibility query with two message names.

It is guaranteed that the descriptor is valid, in other words, no two messages have the same name, no two fields in a message have the same field name or tag number, and each message type in a field must be one of the existing message names.

There will be at most 1000 messages in the descriptor, and each message will have at most 16 fields.

All tokens of the input are separated by spaces.

Output

For each query, output one line containing "Wire-format compatible." (quotes for clarity) if the two messages in this query are wire-format compatible, or otherwise, "Wire-format incompatible." (quotes for clarity)

The sample input/output will be on the next page.

Examples

standard input	standard output

18
message Test1 {
  optional string field = 1 ;
}
message Test2 {
  optional string field_string = 1 ;
}
message Test3 {
  optional string field = 2 ;
}
message Test4 {
  required string field = 1 ;
}
message StringMessage {
  optional string field = 1 ;
}
message Test5 {
  optional StringMessage field = 1 ;
}
4
Test1 Test2
Test1 Test3
Test1 Test4
Test1 Test5
	Wire-format compatible.
	Wire-format incompatible.
	Wire-format incompatible.
	Wire-format incompatible.

5 message A { optional B nest = 1 ; } message B { optional C nest = 1 ; } message C { } message D { optional E nest = 1 ; } message E { } 2 B D A D	Wire-format compatible. Wire-format incompatible.
3 message A { optional A nest = 1 ; } message B { optional C nest = 1 ; } message C { optional B nest = 1 ; } 3 A B A C B C	Wire-format compatible. Wire-format compatible. Wire-format compatible.

Note

In the first example:
For Test1 and Test2, though the two messages have different field names, the serialized message just cares about their field numbers.
For Test1 and Test3, the same field name as the two messages have, their field numbers do not match.
For Test1 and Test4, obviously required is incompatible with optional.
For Test1 and Test5, not all valid UTF-8 strings are valid messages, and vice versa.