One of PA1 teachers is a great romantic. As a result, he only dates girls on full moons. This task will help him optimize his date calendar. The task is to implement functions (not a whole program, just functions) which handle full moon computations. The functions are:

`int isFullMoon ( int y, int m, int d )`
: The function decides whether the date refers to a full moon, or not. The parameters are year, month, and day to test. The result is value 1 (full moon on that day), 0 (no full moon on that day), or `INVALID_DATE` if the input date is invalid.

`int prevFullMoon ( int y, int m, int d, int *prevY, int * prevM, int * prevD )`
: The function computes the full moon day that precedes the given date. The input parameters are year, month, and day of the date in question. Output parameters are `prevY`, `prevM`, and `prevD`, the function fills the parameters with the date of the previous full moon. Return value is either 1 (success) or `INVALID_DATE` (failure, invalid input date). If the function detects an invalid input date, it must return `INVALID_DATE` and must not modify the output parameters (`prevY`, `prevM`, and `prevD`) in any way.

`int nextFullMoon ( int y, int m, int d, int *nextY, int * nextM, int * nextD )`
: The function computes the full moon day that follows the given date. The input parameters are year, month, and day of the date in question. Output parameters are `nextY`, `nextM`, and `nextD`, the function fills the parameters with the date of the next full moon. Return value is either 1 (success) or `INVALID_DATE` (failure, invalid input date). If the function detects an invalid input date, it must return `INVALID_DATE` and must not modify the output parameters (`nextY`, `nextM`, and `nextD`) in any way.

`INVALID_DATE`
: This is a symbolic constant defined in the testing environment. The above functions do use this constant to indicate a failure (invalid date in input parameters).

The input parameters must define a valid date. To be valid, the input must satisfy the following restrictions:

- year must be greater than or equal to 2000 (i.e. all dates before 2000-01-01 are invalid),
- month is valid (1 to 12),
- day is valid (1 to the number of days in the month).

Submit a source file with the implementation of all required functions. Further, the source file must include your auxiliary functions which are called from the required functions.The function will be called from the testing environment, thus, it is important to adhere to the required interface. Use the attached sample code as a basis for your development, complete the required functions and add your required auxiliary functions. There is an example `main` with some test in the attached code. These values will be used in the basic test. Please note the header files as well as `main` is nested in a conditional compile block (`#ifdef/#endif`). Please keep these conditional compile block in place. They are present to simplify the development. When compiling on your computer, the headers and `main` will be present as usual. On the other hand, the header and `main` will "disappear" when compiled by Progtest. Thus, your testing `main` will not interfere with the testing environment's `main`.

Your function will be executed in a limited environment. There are limits on both time and memory. The exact limits are shown in the test log of the reference. The time limits are set such that a reasonable implementation of the naive solution passes all mandatory tests. Thus, the naive solution may be awarded nominal 100%. The algorithm must be improved to pass the bonus test and gain the extra points. Dates very far in the future are tested in the bonus tests (years orders of magnitude greater than 4000).

Advice:

- Download the attached sample code and use it as a base for your development.
- The `main` function in your program may be modified (e.g. a new test may be included). The conditional compile block must remain, however.
- We assume standard Gregorian calendar when counting days. Thus, there is a fixed number of days (30/31) in a month, with the exception of February. February is either 28 days (non-leap year) or 29 days (a leap year). The leap year rules of Gregorian calendar are:

  1. years are not leap years in general,
  2. except multiples of 4 which are leap years,
  3. except multiples of 100 which are not leap years,
  4. except multiples of 400 which are leap years,
  5. except multiples of 4000 which are not leap years.

  Thus, years 2001, 2002, 2003, 2005, ... are not leap years (rule #1), years 2004, 2008, ..., 2096, 2104, ... are leap years (rule #2), years 2100, 2200, 2300, 2500, ... are not leap years (rule #3), years 2000, 2400, ..., 3600, 4400, ... are leap years (rule #4), and years 4000, 8000, ... are not leap years (rule #5).

- Both `prevFullMoon` and `nextFullMoon` functions return a date of the preceding/next full moon. This applies even if the actual parameters form a date with full moon - the output date will be the previous/next full moon, i.e. the output date is never the same as the input date.
- The computation may be based on the fact that the (mean) full moon period is 29.53059027 days, i.e. 29 days 12:44:03. Next, we will use a reference full moon on July 16, 2000 at 13:55:12 UTC. Further, we assume all parameters (day, month, and year) are in UTC.
- The actual time between two full moons is not constant, it oscillates based on the actual position of the Moon, Earth and Sun. The period varies between 29.18 days and 29.93 days (Wikipedia, another explanation). Thus the precise computation is much more difficult, we will not consider the oscillation in this homework. The simplification, however, causes small differences between the results of our functions and the actual lunation dates (e.g. listed on the Internet). This is shown even in the test data. The full moon of December 2019 will actually be on December 12, early in the morning, however, our simplified implementation computes the full moon on December 11, late at night. Next, if you are going to use the published lunation dates, make sure you convert the dates into UTC (the published dates are often in PST, EST, or in local time zone - CET/CEST).

- It is difficult to handle 3-tuples like (year, month, day). It is a good idea to convert the 3-tuple into some other representation, e.g. into a single integer.
- There will be a lot of computation done twice in the program: for the previous and the next full moon computation. Thus, it is a good idea to prepare auxiliary functions and call them twice.
- The years in the mandatory tests do not exceed 2200.
- There is macro `assert` used in the example `main` function. If the value passed to `assert` is nonzero (true), the macro does nothing. On the other hand, if the parameter is zero, the macro stops the execution and reports line, where the test did not match (and shall be fixed). Thus, the program ends silently when your implementation passes the tests correctly.