ProgTest ▶ BIE-PA1 (22/23 ZS) ▶ Homework 03 ▶ Tower clock **Logout**

Tower clock

Submission deadline:	**2022-11-14 11:59:59**	1388197.465 sec
Late submission with malus:	**2022-12-31 23:59:59** (Late submission malus: 100.0000 %)	
Evaluation:	**0.0000**	
Max. assessment:	**5.0000** (Without bonus points)	
Submissions:	0 / 20 Free retries + 10 Penalized retries (-10 % penalty each retry)	
Advices:	0 / 2 Advices for free + 2 Advices with a penalty (-10 % penalty each advice)	

The task is to implement a function (not a whole program, just a function) which computes how many times does a tower clock strike its bells.

We assume a tower clock equipped with two bells. The clock strikes its bells in the following pattern:

- bell #1 indicates minutes. The clock strikes it once, twice, three times and four times at XX:15, XX:30, XX:45 and XX:00, respectively,
- bell #2 indicates hours. At the top of the hour, the bell is struck 1 to 12 times, based on the hour. Thus, at midnight (0:00] - 12 times, at 1:00 - once, at 2:00 - twice, ..., at 12:00 - 12 times, at 13:00 - once, at 14:00 - twice, ...,
- both bells are disabled on Sundays. That is, the clock strikes its bells on Saturday at 23:45 and then on Monday at 0:00.

Our function will be given two timestamps - the start and the end of the time interval. Each timestamp is defined by year, month, day, hour, and minute. The function computes the strikes foe either bell in the interval. We assume the boundaries inclusive, i.e., if the bells are struck at the moment of the start/end, these are included.

The required interface is:

```
int bells ( int y1, int m1, int d1, int h1, int i1,
            int y2, int m2, int d2, int h2, int i2,
            long long int * b1, long long int * b2 )
```

`y1, m1, d1, h1, i1`
: are input parameters representing year, month, day, hour, and minute. They define the start of the interval,

`y2, m2, d2, h2, i2`
: are input parameters representing year, month, day, hour, and minute. They define the end of the interval,

`b1`
: is an output parameter. The function must fill it with the total number of strikes bell #1 is struck. The parameter is to be set only if the input parameters are valid. If the input parameters are invalid, the function must not modify the output parameter in any way.

`b2`
: is an output parameter. The function must fill it with the total number of strikes bell #2 is struck. The parameter is to be set only if the input parameters are valid. If the input parameters are invalid, the function must not modify the output parameter in any way.

return value
: must be set to 1 if the input was valid, 0 if the input parameters were invalid.

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

- year must be greater or equal to 1600,
- month is valid (1 to 12),
- day is valid (1 to the number of days in the month),
- hour is valid (0 to 23),
- minute is valid (0 to 59),
- the end timestamp must not precede the start timestamp.

Submit a source file with the implementation of the required function `bells`. Further, the source file must include your auxiliary functions which are called from `bells`. The function will be called from the testing environment, thus, it is important to adhere to the required interface. Use the sample code below as a basis for your development, complete `bells` and add your required auxiliary functions. There is an example `main` with some test in the sample below. 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`.

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. This rule is not officially adopted. There are proposals to add this rule to the definition of Gregorian calendar; the calendar would approximate Earth movement more precisely with this additional rule. Nevertheless, our implementation considers this rule.

Thus, years 1901, 1902, 1903, 1905, ... are not leap years (rule #1), years 1904, 1908, ..., 1996, 2004, ... are leap years (rule #2), years 1700, 1800, 1900, 2100, ... are not leap years (rule #3), years 1600, 2000, 2400, ..., 3600, 4400, ... are leap years (rule #4), and years 4000, 8000, ... are not leap years (rule #5).

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 correct implementation of the naive solution passes all mandatory tests. Thus, the solution may be awarded nominal 100% percent. The algorithm must be improved to pass the bonus test and award the extra points. There are long intervals tested in the bonus tests (years are many orders of magnitude greater than 4000).

Example calls of the function are included in the attached archive.

Advice:

- Copy the sample code from the archive and use it as a base for your development.
- The main in your program may be modified (e.g. a new test may be included). The conditional compile block must remain, however.
- It is difficult to handle 5-tuples like (year, month, day, hour, minute). For instance, it is awkward to compare them. It is a good idea to convert the 5-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 start and end timestamps. Thus, it is a good idea to prepare auxiliary functions and call them twice.
- The years in the mandatory tests do not exceed 4000.
- The naive algorithm iterates over all timestamps the clock strikes and sums the strikes. However, this is very inefficient and may not pass the time limits. Instead, accumulate the number of strikes over longer intervals, e.g., iterate over whole days or even longer time intervals.
- The output values are extremely high for long intervals in the bonus tests. It may exceed the limits of the `int` data type. Therefore, the output parameter is `long long int`.
- The attached source lists the header files available in the testing environment. Other header files are not present, moreover, you cannot include them by means of `#include` directives in your submitted code. Please note that `time.h` is not present.

Sample data: Download

Submit: Choose File Submit

Reference