1. **Initial Thoughts & Approach**
    a. **Approach A:** Originally I thought of a very simple solution to the problem. Assuming that the users correctly report their first payday. Using this payday, I can check if the date to query is a payday for that user by doing module 14 of the number of days b/w first payday and the date to query. If the remainder is 0, it is a payday.

       And given a date to query, I can calculate the next payday for that user by:
       i) n = (date_to_query - payday) / 14
       ii) Round up 'n'
       iii) next payday = first payday + (n x 14) days

    b. **Approach B:** Another approach I thought of was to have a helper function that can give the difference between 2 dates in number of weeks and days (as remainders). So Something like- LAST_KNOWN_PAY_DATE minus DATE_TO_QUERY and get the number of weeks between the 2 dates. If the number is a whole number, then it falls directly on a weekly interval. If the number has a remainder then it's a few days off and not a payday. If the number is a whole number and even, then it's potentially a bi-weekly payday, as long as the DATE_TO_QUERY is not in the given list of holidays.

2. I started brainstorming real-world scenarios and came across various edge cases for which the above 2 approaches will not work.
   Following is my thought process as I was analyzing the problem and trying to find a solution that solves the issues in the above approach 'A' and 'B'
    a. The approach 'B' works perfectly fine with the BI-WEEKLY pay cycle but, will not work if the same logic is used for the WEEKLY pay cycle. For example, checking if the difference between two dates (in weeks) is even, clearly tells us that it will not work for the weekly pay cycle.
    b. **Frequency:**
        i. Given a pay cycle, I am setting a frequency to use for calculation.
        ii. BI-WEEKLY is every two weeks, so I am using 14 days as a frequency placeholder.
        iii. I am leveraging python's 'timedelta' function to avoid running in simple cases like leap year and the different number of days in a month.
    c. **First payday:**
        i. I am assuming this will be accurately reported by the user.
        ii. Can be used to calculate the next consecutive paydays by incrementing by 2 weeks / 14 days.
        iii. Can be used to reject the date to query if it is before the first payday of the user. For the sake of simplicity, I will ignore this rejection of the first payday.

    iv. If the date to query is less than the first payday, I have a mechanism of going backward by subtracting the frequency. Similarly, if the date to query is greater than the first payday, I am going forward by adding the frequency.

d. **Last payday:**
    i. The latest date when the user got paid
    ii. Initially, I decided to include the last payday of the user to optimize the calculations. How will this optimize the solution? Basically, pick the starting point of calculation whichever is near to the date to query.
    iii. I thought that the last payday can be cached in the user model and passed into the library to perform any calculation. But, from a business point of view, we don't want the user to keep reporting their last payday. On the other hand, the last payday is something that can be maintained by Perpay, this will require extra processing, however.
    So, given the scope of this project and for the sake of simplicity, I decided to ignore the last payday for now.

e. **Holidays**:
    i. One of the very important real-world cases would be to incorporate the holidays.
    ii. As I mentioned before in approach A that I can just do modulo 14 to check if a date to query is a payday or not.
    However, given that a user gets paid bi-weekly on Fridays and considering the following real-world edge case, we will get wrong answers with approach A.
        1. Given first payday = Feb 7th, 2020 (Friday)
        Date to check = Dec 25th, 2020 (Friday) ← A Holiday
        There are 322 days in between
        322%14 = 0 tells us that it is a payday which is wrong because Dec 25th, 2020 is a holiday (Christmas).

        2. Similarly, If the given payday is Dec 11th, 2020 (Friday)
        To find the next payday, adding 14 days will give us Dec 25th, 2020 (Friday) which is a Holiday. Hence, invalid.
        Moreover, we can not just keep adding 14 days to get the next paydays because of leap years. Therefore, taking holidays into account is very essential.

    iii. Perpay deals with users with different pay cycles. Hence, in addition to the common holidays (New year and Christmas, etc), a user can report other holidays that are only applicable to them. This holidays logic is scaleable to different types of pay cycles. Therefore, I am assuming that the user model will contain the holidays that are applicable to that user. And I can incorporate those holidays while making calculations. How?

**Solution:**

So, in order to tackle holidays and making sure they are handled as I encounter them, I start from the first_payday and keep adding 14 days until I get close to the ‘date to query’. While I am adding 14 days, I am checking against the provided holidays and updating the current payday.

How am I updating the current payday if it is a holiday?

**Dealing with Holidays**:

- If a user gets paid on Wednesdays but, due to a holiday on Wednesday the user’s payday will be the next day i.e Thursday.
- Another case is if a user gets paid on Fridays but, due to a holiday on Friday the user’s payday will be the day before i.e. Thursday instead of Monday(next week).

**Possibility of further optimization:**

One optimization is, instead of incrementing by 2 weeks (bi-weekly) to get close to far-off dates, I could go ~year-by-year or ~month-by-month depending on how far the date to query is.

Another Optimization is caching paydays. Since we are adding 14 days (and incorporating holidays) in a loop, we can optimize the calculation by starting with a payday that is closer to the ‘date to query’. In order to achieve it, we need to cache the calculated paydays somewhere. Moreover, the cached paydays can be used to get the next x paydays of a user.

In terms of deciding between space and runtime in this scenario, I would lean towards getting the best time complexity because processing is very expensive as compared to storage.

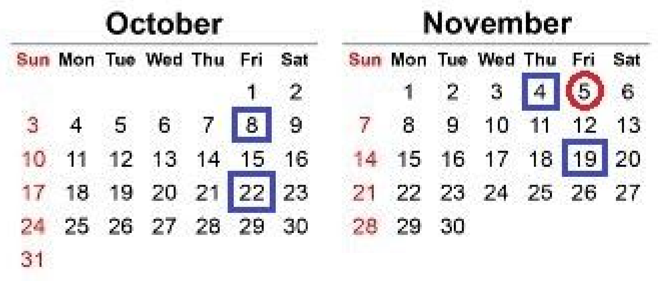
For the sake of simplicity, I decided to stick to the approach I described in the above, **Solution,** paragraph.

f. **Default payday**:

   i. It is the weekday when the user normally gets paid (MONDAY -> FRIDAY)

   ii. I decided to include the default payday because it will help us incorporate the holiday logic by resetting the payday to its default payday cycle. How?

While adding 14 days in a loop, I have a mechanism that if I land on a payday and if it was a holiday then my holiday logic will update the payday (as described in the **Solution** paragraph above. However, in the next iteration (when we add 14 days again), we need to make sure that the payday is set to the default payday of the user otherwise it will mess up the next payday cycle. For Example:



October

| Sun | Mon | Tue | Wed | Thu | Fri | Sat |
|---|---|---|---|---|---|---|
| | | | | | 1 | 2 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | | | | | | |

November

| Sun | Mon | Tue | Wed | Thu | Fri | Sat |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | | | | |

- Assume a user gets paid on Fridays (as shown above)
- The user gets paid on Oct 22nd
- +14 days will give us Nov 5th which is a holiday.
- So, my logic changes the current payday to Nov 4th (the day before) and stores it (if needed).
- Now, in the next iteration, if I add 14 days again to get the next payday, it will be Nov 18th (Thursday) which is wrong. It should be Nov 19th (Friday).
- That is when the default payday is used. In order to get Nov 19th, before I calculate the next payday, I check if the current payday follows the default pay cycle. In the example above, it does not, so before I add 14 days, I reset Nov4th to Nov5th.

I hope this example demonstrates the edge case I am trying to tackle. This way of incorporating holidays will work for different pay cycles and different holidays.

**NOTES:**

1. To establish test cases, I am using example calendars inside “*Example Calendars.pdf*”
2. Running Tests:

   Run “ *$docker build .* “ inside the submission directory.

   Or install the packages inside requirements.txt
   And run *“ $python3 -m unittest discover -v -s ./test_suite -p "*_test.py "* from the submission directory.