Part I : Conceptual Problems

Problem 1:

We have 52 week price list for a stock S. Suppose that during this period, we want to buy 100 stocks of S and sell all of them at a later day (within the 52 week window). We want to know when should we have bought and sold in order to make maximum profit. (If there is no profit making scenario, then it should be reported).

Example, for a small 4 week window:
P1 = 14, P2 = 6, P3 = 7, P4 = 11
Then the output should be "Buy on 2 and sell on 4". Short selling is not allowed.

Question: How can you best classify the type of the algorithm? (Greedy or Divide & Conquer or Dynamic Programming, etc..)

Question 2: Give the time complexity of the algorithm. Briefly explain your answer.

Problem 2:

Given a list of stocks with their respective market capitalizations. The stock's market cap are updated every minute based on its trading in the market. I have a strategy that wants a list of top 10 highest capped stocks in the market. This query can happen multiple times during the day.

Question: What would be the most appropriate data structure that can be used to efficiently implement the list of stocks? Discuss pros/cons of various data structures. Please briefly explain your answer.

Problem 3:

What would be the most appropriate way to implement a stack and a queue **together** efficiently, in a single data structure. The number of elements is infinite. The retrieval and insertion should both happen in constant time. Discuss pros/cons of various data structures.

Problem 4:

What are the basic difference between using ArrayList and LinkedList as a high volume data insertion, deletion and updation activities?

Part II: Programming

Problem 1:

Introduction: There is a company that generates its own internal unique identifier for each security in the market. This unique identifier is called "S2ID". The company stores the mapping from the S2ID to various well known ID's (such as Sedol, Cusip, ISIN etc.) in flat files. **(IDMapping.txt)**
The file also mentions the various changes in the ID mapping occurring since the inception of the company, i.e. the complete history since the start is present in the file. This is done by mentioning a start and an end date for each ID. An ID is said to be "active" during this period.
It has been observed that a particular well known ID can be mapped to different S2ID during different times.
For example, consider Sedol: 2792134. (You can refer to the input file for better understanding)
During 19961231 to 19970731, the sedol is mapped to ARGADU1.
During 19970801 to 19981130, the sedol is mapped to ARGADU2.
During 19981201 to 19981231, the sedol is mapped again to ARGADU1.
During 19990101 to 20080911, the sedol is mapped again to ARGADU2.

There is another file named as “AssetIdentity” which stores the detailed description of each of the S2ID along with the changes to the ID since the inception of the company. One important information in this file is the mapping from S2ID to “RootID” which is a company identifier. There is a many-to-many relationship between a RootID and a S2ID at various points in time.

There is a frequent requirement to find the list of various well known ID’s corresponding to a RootID, given a particular date. The returned list of well known ID’s should be active at the specified date.
Write a script that helps user to get a list of well known ID’s for a particular RootID and date.

Input: The program asks user to choose between: Sedol/Cusip/ISIN, then takes as input:

1. A RootID
2. The date for which to check (in YYYYMMDD format)

Output: A comma separated list of unique ID’s

The code must take the input from stdin and produce output to stdout as specified above.

Problem 2:

Introduction: A researcher wants an API that iteratively keeps returning numbers in the Fibonacci series. Write a class will behave like an Iterator, that will expose a next() function. On each call to next(), the next integer in the Fibonacci series should be returned.
The class will also provide a constructor with an integer argument **'n'**. The argument denotes the "jump" value that will initialize the object 'n' steps ahead from the start of the series. The call to next() should then return the values from this stepped ahead index in the series.
Example:
If an object is initialized as:

```
FibIterator iterator = new FibIterator(4);
int next = iterator.next();
```

The value of *next* should be 5.

Input: None

Output: The API should expose the next() function, that will return the appropriate number.