Algorithms and Advanced Programming (AAP) - CA

Part 1:1

Using Stock.csv dataset as last digit of student number is 2.
Sorting algorithm based on column 5 as second last digit of student number is 3.
Column 5 in Stock.csv is “buy_date”.

I chose to use merge sort due to its quick sorting time. It uses the divide and conquer approach where it keeps dividing the dataset by 2 giving it a time complexity of O(NlogN).

Input: ArrayList , start , end indexes of the ArrayList
Processing: Sort items in the ArrayList based on buy_date -> falling back to stockNo if date equal
Output: Sorted ArrayList

```java
//Part1.1 - sorting algorithm - merge sort
static void merge(ArrayList<Stock> myStock, int lowerB, int mid, int upperB){
    int i = lowerB;
    int j = mid;
    //use temp array to store merged sub-sequence
    int t = 0;
    Stock[] temp = new Stock[upperB - lowerB];
    while(i < mid && j < upperB){
        //compareTo has been overridden to compare based on column 5 (buyDate) and then compare by stockNo if dates
        //are equal
        int compare = myStock.get(i).compareTo(myStock.get(j));
        if(compare < 0){
            temp[t] = myStock.get(i); i++; t++;
        }
        else{
            temp[t] = myStock.get(j); j++; t++;
        }
    }
    //tag on remaining sequence
    while(i < mid){
        temp[t] = myStock.get(i); i++; t++;
    }
    while(j < upperB){
        temp[t] = myStock.get(j); j++; t++;
    }
    //copy temp back to f
    i = lowerB;
    t = 0;
    while(t < temp.length){
        myStock.set(i, temp[t]);
        i++; t++;
    }
}
```

```java
static void mergeSort(ArrayList<Stock> myStock, int lowerB, int upperB){
    if(lowerB + 1 < upperB){
        int mid = (lowerB + upperB)/2;
        mergeSort(myStock, lowerB, mid);
        mergeSort(myStock, mid, upperB);
        merge(myStock, lowerB, mid, upperB);
    }
}
```

Because I am sorting based on “buy_date” I had to override the compareTo method to of the Stock class to compare using the “buy_date” field and if this field was equal then to compare based on the “stockNo” field.

```java
@Override
public int compareTo(Object obj) {

    /*
    Based on second last number of my student number being 3 - search will be done on colum 5 (buyDate)
    */
    Stock stk = (Stock)obj;
    //check if date is after
    if ( this.buyDate.toInstant().isAfter(stk.getBuyDate().toInstant())){
        return 1;
    }
    //check if date is before
    else if ( this.buyDate.toInstant().isBefore(stk.getBuyDate().toInstant())){
        return -1;
    }
    //if dates are equal, fallback to sorting by stockNo
    else{
        return stockNo - (stk.stockNo);
    }
}
```

To show the data was actually sorted I print a sub-section of the data(first 10) before running the sort method and then print the same sub-section after sort. As can be seen below the data is not sorted by the 5th column – buy_date.

```
/Users/colin.cumings/Library/Java/JavaVirtualMachines/openjdk-17.0.2/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=56580:/Applications/IntelliJ IDEA CE.app/Contents/bin -Dfile.encoding=
Part1.Stock [StockNo=1, Part1.Stock size=5.46, Cost=835.85, Product Type=food, Buy date=Wed Apr 24 00:00:00 IST 2019, Product name=roll]
Part1.Stock [StockNo=2, Part1.Stock size=6.33, Cost=381.99, Product Type=clothing, Buy date=Sun Oct 16 00:00:00 IST 2016, Product name=polo]
Part1.Stock [StockNo=3, Part1.Stock size=1.4, Cost=811.99, Product Type=furniture, Buy date=Sat Apr 15 00:00:00 IST 2017, Product name=desk]
Part1.Stock [StockNo=4, Part1.Stock size=0.65, Cost=283.01, Product Type=clothing, Buy date=Wed May 06 00:00:00 IST 2020, Product name=tie]
Part1.Stock [StockNo=5, Part1.Stock size=4.54, Cost=819.67, Product Type=furniture, Buy date=Mon Dec 31 00:00:00 GMT 2018, Product name=desk]
Part1.Stock [StockNo=6, Part1.Stock size=4.17, Cost=698.14, Product Type=clothing, Buy date=Mon Jan 28 00:00:00 GMT 2019, Product name=jeans]
Part1.Stock [StockNo=7, Part1.Stock size=0.82, Cost=785.34, Product Type=food, Buy date=Mon Mar 30 00:00:00 IST 2020, Product name=noodles]
Part1.Stock [StockNo=8, Part1.Stock size=5.62, Cost=411.26, Product Type=clothing, Buy date=Thu Jun 03 00:00:00 IST 2021, Product name=socks]
Part1.Stock [StockNo=9, Part1.Stock size=9.7, Cost=30.0, Product Type=furniture, Buy date=Mon May 17 00:00:00 IST 2021, Product name=cot]
Part1.Stock [StockNo=10, Part1.Stock size=6.79, Cost=412.61, Product Type=clothing, Buy date=Sat Nov 12 00:00:00 GMT 2016, Product name=shirt]
Sorting the data took: 20370
Part1.Stock [StockNo=986, Part1.Stock size=5.13, Cost=242.28, Product Type=clothing, Buy date=Thu Jan 01 00:00:00 GMT 2015, Product name=tie]
Part1.Stock [StockNo=1271, Part1.Stock size=5.39, Cost=293.74, Product Type=furniture, Buy date=Thu Jan 01 00:00:00 GMT 2015, Product name=shelves]
Part1.Stock [StockNo=1319, Part1.Stock size=8.26, Cost=472.77, Product Type=furniture, Buy date=Thu Jan 01 00:00:00 GMT 2015, Product name=dining table]
Part1.Stock [StockNo=1493, Part1.Stock size=4.35, Cost=590.89, Product Type=clothing, Buy date=Thu Jan 01 00:00:00 GMT 2015, Product name=tie]
Part1.Stock [StockNo=1993, Part1.Stock size=8.33, Cost=260.02, Product Type=furniture, Buy date=Thu Jan 01 00:00:00 GMT 2015, Product name=sofa]
Part1.Stock [StockNo=3027, Part1.Stock size=3.66, Cost=770.61, Product Type=food, Buy date=Thu Jan 01 00:00:00 GMT 2015, Product name=sauce]
Part1.Stock [StockNo=8368, Part1.Stock size=6.05, Cost=62.32, Product Type=clothing, Buy date=Thu Jan 01 00:00:00 GMT 2015, Product name=dress]
Part1.Stock [StockNo=9344, Part1.Stock size=1.85, Cost=380.37, Product Type=clothing, Buy date=Thu Jan 01 00:00:00 GMT 2015, Product name=t-shirt]
Part1.Stock [StockNo=2502, Part1.Stock size=5.47, Cost=852.88, Product Type=furniture, Buy date=Fri Jan 02 00:00:00 GMT 2015, Product name=desk]
Part1.Stock [StockNo=4164, Part1.Stock size=3.33, Cost=454.28, Product Type=food, Buy date=Fri Jan 02 00:00:00 GMT 2015, Product name=bread]
```

Part1:2

Analysing the time complexity of the sorting algorithm.
To do this I used System.nanoTime() and the divided by 1000 to give me a value in microseconds so that the numbers were easier to work with.
I also ran each test 3 times to get an average for the time.

```
//Part1.1 - merge sort
startTime = System.nanoTime();  //record start time just before we execute the sorting algorithm
mergeSort(myStock, lowerB: 0, upperB: 10000);  //myStock.length can be substituted with 10, 100, 1000, 5000 for time complex analysis
endTime = System.nanoTime();  //record end time just after sorting algorithm runs

//Part1.2 - time complexity measure in microseconds for visibility
totalTime = (endTime - startTime)/1000;
System.out.println("Sorting the data took: " + totalTime);
```

Number of records to sort	Microsecond's taken for algorithm to run
10	(299 +326+245)/3 = 290
100	(1245+1214+1284)/3 = 1248
1000	(3945 + 4135 + 3923)/3 = 4001
5000	(10033 + 9561 + 10227)/3 = 9930
10000	(25518 + 26011 +25212)/3 = 25580

As we can see from the above data the time taken to run the sort does not increase in line with the number of records:
290 to sort 10 records
25580 to sort 10000 records
While the number of records has increased 1000x, the time to sort the records has only increased 88x.
This is due to the O(NlogN) time complexity nature of the merge sort algorithm.

Part1:3

Searching algorithm based on column 5 – buy_date.
Chose binary search as the search algorithm due to its average time complexity of O(logN).

Input: Sorted ArrayList, date to search for
Processing: Search for items in list that match the date
Output: Print to screen "Not an existing date" if no items in list with that date; or list all items with that date

To be able to provide all items in the ArrayList bought on a particular date I wrote 2 search algorithms, one to find the first index in the list of the searched date and a second to find the last index. Then I could list all items between these indexes to show all items bought on the searched date.

Search first

```java
static int searchFirst(ArrayList<Stock> myStock, String date) throws ParseException {
    // convert search date string to Date
    SimpleDateFormat enteredDate = new SimpleDateFormat( pattern: "yyyy-MM-dd");
    Date searchDate = enteredDate.parse(date);
    //System.out.println("Searching for date: " + searchDate);
    int first = 0;
    int last = myStock.size() -1;
    //for analysing time complexity we can adjust the amount on entries we search by adjusting this variable
    //int last = 10;
    int result = -1;
    while(first <= last){
        int middle = (first +last)/2;
        Date stockDate = myStock.get(middle).getBuyDate();
        //System.out.println("Checking if " + stockDate + " is after " + searchDate );
        if (stockDate.before(searchDate)){
            //System.out.println("Confirmed that " + stockDate + " is after " + searchDate );
            first = middle +1;
        }
        else if (stockDate.after(searchDate)){
            //System.out.println("Confirmed that " + stockDate + " is before " + searchDate );
            last = middle -1;
        }
        else{
            result = middle;
            //if middle is the same as searchDate then move left to find first occurrence
            last = middle -1;
        }
    }
    return result;
}
```

Search last

```java
//return the last index of the
static int searchLast(ArrayList<Stock> myStock, String date) throws ParseException {
    // convert search date string to Date
    SimpleDateFormat enteredDate = new SimpleDateFormat( pattern: "yyyy-MM-dd");
    Date searchDate = enteredDate.parse(date);
    //System.out.println("Searching for date: " + searchDate);
    int first = 0;
    int last = myStock.size() -1;
    int result = -1;
    while(first <= last){
        int middle = (first +last)/2;
        Date stockDate = myStock.get(middle).getBuyDate();
        //System.out.println("Checking if " + stockDate + " is after " + searchDate );
        if (stockDate.before(searchDate)){
            //System.out.println("Confirmed that " + stockDate + " is after " + searchDate );
            first = middle +1;
        }
        else if (stockDate.after(searchDate)){
            //System.out.println("Confirmed that " + stockDate + " is before " + searchDate );
            last = middle -1;
        }
        else{
            result = middle;
            //if middle is the same as searchDate then move right to find first occurrence
            first = middle +1;
        }
    }
    return result;
}
```

Print output to screen based on searches

```
if(firstResult == -1){
    //If no items found on date searched
    System.out.println("Not an existing buy_date!");
}
else{
    //only search for last index if there was a result for first index search
    int lastResult = searchLast(myStock, dateToSearch);
    System.out.println("Date first found at index: " + firstResult);
    System.out.println("Date last found at index: " + lastResult);
    System.out.println("Below is the list of stock bought on: " + dateToSearch);
    //print out all items bought on the date searched
    for(int num = firstResult; num <= lastResult; num++ ){
        System.out.println(myStock.get(num));
    }
}
```

Example output of running this search:
Date not found:

```
Please enter date to search in the following format: yyyy-mm-dd
1999-01-01
Not an existing buy_date!
```

Dates found:

```
Please enter date to search in the following format: yyyy-mm-dd
2015-12-12
Date first found at index: 1396
Date last found at index: 1399
Below is the list of stock bought on: 2015-12-12
Part1.Stock [StockNo=181, Part1.Stock size=9.66, Cost=974.24, Product Type=food, Buy date=Sat Dec 12 00:00:00 GMT 2015, Product name=sauce]
Part1.Stock [StockNo=2014, Part1.Stock size=8.56, Cost=375.04, Product Type=furniture, Buy date=Sat Dec 12 00:00:00 GMT 2015, Product name=desk]
Part1.Stock [StockNo=4134, Part1.Stock size=9.84, Cost=958.92, Product Type=clothing, Buy date=Sat Dec 12 00:00:00 GMT 2015, Product name=dress]
Part1.Stock [StockNo=7973, Part1.Stock size=7.82, Cost=97.58, Product Type=food, Buy date=Sat Dec 12 00:00:00 GMT 2015, Product name=quinoa]
```

Part1:4

Analysing the time complexity of the searching algorithm.
To do this I used System.nanoTime() and the divided by 1000 to give me a value in microseconds so that the numbers were easier to work with.
I also ran each test 3 times to get an average for the time.

```
startTime = System.nanoTime();  //record start time just before we execute the searching algorithm
firstResult = searchFirst(myStock, dateToSearch);
endTime = System.nanoTime();  //record end time just after searching algorithm runs
```

```
//Part1.4 - time complexity measure for search algorithm in microseconds for visibility
totalTime = (endTime - startTime)/1000;
System.out.println("Searching the first occurrence of the date  took: " + totalTime);
```

To adjust the number of records searched for time complexity analysis I adjusted the size of the array to search as seen in below code:

```java
static int searchFirst(ArrayList<Stock> myStock, String date) throws ParseException {
    // convert search date string to Date
    SimpleDateFormat enteredDate = new SimpleDateFormat( pattern: "yyyy-MM-dd");
    Date searchDate = enteredDate.parse(date);
    //System.out.println("Searching for date: " + searchDate);
    int first = 0;
    int last = myStock.size() -1;
    //for analysing time complexity we can adjust the amount on entries we search by adjusting this variable
    //int last = 10;
```

Number of records to search	Microsecond's taken for algorithm to run
10	(222 +350 + 228)3 = 262
100	(212 + 347 + 230)/3 = 263
1000	(292 + 271 + 214)/3 = 259
5000	(277 + 326 + 299)/3 = 301
10000	(427 + 220 + 225)/3 = 290

As we can see from the above data the time taken to run the serach does not increase in line with the number of records:
Our times are nearly constant which suggests our date searched during the test could be very close to the middle of the sorted ArrayList in the larger numbers of search records in the test. The average and worst time complexity we expect from binary searching is O(logN) which is still efficient.

Part2:1

For this question I assumed the input for the new record would be via the terminal and that it would be input one field at a time by the user.

```java
//Part2.1 - accept new record and add it at the end of the arraylist
//Example Input for stock: (12.2, 120.3, "furniture", 2015-01-10, "L-Sofa")
//product_size,cost,product_type,buy_date,product_name

//get user input
Scanner myInput = new Scanner(System.in);
System.out.println("Entry of new stock item....");
System.out.println("Please enter the product size: ");
float size = myInput.nextFloat();
System.out.println("Please enter the product cost: ");
float cost = myInput.nextFloat();
//bool to flag when to exit while loop
boolean validType = false;
String type = "";
//loop until user enter correct type
while(validType == false) {
    System.out.println("Please enter the product type: ");
    type = myInput.next();
    //Part2.3- exception handling if invalid input entered
    try {
        AllowedTypes.valueOf(type);
        validType = true;
    } catch (IllegalArgumentException err) {
        System.out.println(err);
        System.out.println("Product type can only be one of : furniture | clothing | food");
    }
}
System.out.println("Please enter the date the product was bought: ");
String buyDate = myInput.next();
```

Part2:2

To validate the date entered by the user I wrote a method to check the date validity called checkDate:

Input: Date
Processing: Check is date is between start date and todays date
Ouput: Boolean true or false

- I used Date() to get todays date and hard coded the start date of the company as that will not change

```java
////Part2.2 - method to validate date input
static boolean checkDate(String date) throws ParseException {
    //get current date
    Date currentDate = new Date();
    //convert input string to date object
    Date inputDate = new SimpleDateFormat( pattern: "yyyy-MM-dd").parse(date);

    //check date is not after current date
    if(inputDate.after(currentDate)){
        return false;
    }

    //check date is not before start of company
    String start = "2014-05-01";
    Date startDate = new SimpleDateFormat( pattern: "yyyy-MM-dd").parse(start);
    if(inputDate.before(startDate)){
        return false;
    }
    return true;

}
```

I then used this method when the user entered the buy_date field:

```java
System.out.println("Please enter the date the product was bought: ");
String buyDate = myInput.next();
//Part2.2 - date validation on user input
if (checkDate(buyDate)){
    System.out.println("Please enter the product name: ");
    String name = myInput.next();

    //generate the next available stock number
    int stockNum = myStock.size() +1;
    //create stock object from the user Input data
    Stock userAddedStock = new Stock(stockNum, size, cost, type, ft.parse(buyDate), name);
    //add new stock entry to arraylist
    myStock.add(userAddedStock);

    //check if new entry added
    // show first 10 objects in array after merge sorting
    for (i = 9995; i < myStock.size(); i++) {
        System.out.println(myStock.get(i));
    }
}
else {
    System.out.println("Please enter a valid date between 2014-05-01 and today's date: ");
}
```

Part2:3

Java exception if product type is anything other than the 3 products mentioned:

For cleanliness of code I created enum of AllowedTypes to use in the exception:

```java
//Part2.3- exception handling allowed values
enum AllowedTypes{
    food, furniture, clothing
}
```

Then I placed the exception in a while loop so that we do not continue with data that is not valid

```java
//loop until user enter correct type
while(validType == false) {
    System.out.println("Please enter the product type: ");
    type = myInput.next();
    //Part2.3- exception handling if invalid input entered
    try {
        AllowedTypes.valueOf(type);
        validType = true;
    } catch (IllegalArgumentException err) {
        System.out.println(err);
        System.out.println("Product type can only be one of : furniture | clothing | food");
    }
}
```