Workshop: Basic Algorithms

This document defines the exercise for "Java OOP Advanced" course @ Software University. Please submit your solutions (source code) of all below described problems in Judge

I. Recursion

1. Recursive Array Sum

Write a program that finds the sum of all elements in an integer array. Use **recursion**.

Note: In practice recursion should not be used here (instead use an **iterative solution**), this is just an exercise.

Examples

Input	Output
1 2 3 4	10
-1 0 1	0

Hints

Write a **recursive** method. It will take as arguments the **input array** and the **current index**.

- The method should return the **current element** + the **sum of all next elements.**
- The recursion should stop when there are no more elements in the array.

```
private static int sum(int[] nums, int index){
    // TODO: Set bottom of recursion
    // TODO: Return the sum of current element + sum of elements to the right
}
```

2. Recursive Factorial

Write a program that finds the factorial of a given number. Use **recursion**.

Note: In practice, recursion should not be used here. Instead, you should use an **iterative solution.** This type of solution is for exercise purposes.

Examples

Input	Output
5	120
10	3628800

Hints

Write a **recursive** method. It will take as arguments an integer number.

- The method should return the **current element** + the **result of calculating factorial of current element - 1** (obtained by recursively calling it).

- The recursion should stop when there are no more elements in the array.

```
private static long calculateFactorial(int n){
    // TODO: Set bottom of recursion
    // TODO: Return the multiple of current n and factorial of n
}
```

Greedy Algorithms

3. Sum of Coins*

Write a program, which gathers a sum of money, using the least possible number of coins. This is the **range of possible coin values**:

- **{ 1, 2, 5, 10, 20, 50 }**

You will receive a **desired sum**. The goal is to **reach the sum using as few coins as possible using a greedy approach**. We'll assume that each coin value and the desired sum are **integers**. There is a skeleton, which you can download from Judge. Use the **Main** class in the skeleton.

Examples

Input	Output	Comments
Coins: 1, 2, 5, 10, 20, 50 Sum: 923	Number of coins to take: 21 18 coin(s) with value 50 1 coin(s) with value 20 1 coin(s) with value 2 1 coin(s) with value 1	18*50 + 1*20 + 1*2 + 1*1 = 900 + 20 + 2 + 1 = 923
Coins: 1 Sum: 42	Number of coins to take: 42 42 coin(s) with value 1	
Coins: 3, 7 Sum: 11	Error	Cannot reach desired sum with these coin values
Coins: 1, 2, 5 Sum: 2031154123	Number of coins to take: 406230826 406230824 coin(s) with value 5 1 coin(s) with value 2 1 coin(s) with value 1	Solution should be fast enough to handle a combination of small coin values and a large desired sum
Coins: 1, 10, 9 Sum: 27	Number of coins to take: 9 2 coin(s) with value 10 7 coin(s) with value 1	Greedy approach produces non-optimal solution (9 coins to take instead of 3 with value 9)

Greedy Approach

For this problem, a greedy algorithm will attempt to take the best possible coin value (which is the largest), then take the next largest coin value and so on, until the sum is reached or there are no coin values left. There may be a different amount of coins to take for each value. In one of the examples above, we had a very large desired sum and relatively small coin values, which means we'll need to take a lot of coins. It would not be efficient (and may even cause an Exception) if we return the result as a **`List<Integer>.`** A more practical way to do it is to use a **`Map<Integer, Integer>`**, where the keys are the coin values and the values are the number of coins to take for the specified coin value. Therefore, in the second example (coin values = { 1 }, sum = 42), instead of returning a list with 42 elements in it, we'll return a dictionary with a single key-value pair: 1 => 42.

Greedy Algorithm Implementation

You are given an implemented **`main()`** method with sample data. Your task is to implement the **`chooseCoins()`** method:

```
public static void main(String[] args) {
    int[]  availableCoins = new int[] {1, 2, 5, 10, 20, 50};
    int targetSum = 923;

    Map<Integer, Integer> selectedCoins = ChooseCoins(availableCoins, targetSum);

    System.out.println(String.format("Number of coins to take: %d",
            selectedCoins.values().stream().mapToInt(Integer::intValue).sum()));

    for (Integer key : selectedCoins.keySet()) {
        System.out.println(String.format("%d coin(s) with value %d", selectedCoins.get(key), key));
    }
}

private static Map<Integer, Integer> ChooseCoins(int[] availableCoins, int targetSum) {
    // TODO
    throw new IllegalArgumentException();
}
```

Since at each step we'll try to take the largest value we haven't yet tried, it would simplify our work to order the coin values so we can iterate them in descending order:

```
Arrays.sort(availableCoins);
List<Integer> sortedCoins  = Arrays.stream(availableCoins).boxed().collect(Collectors.toList());
Collections.reverse(sortedCoins);
```

Now, taking the largest coin value at each step is simply a matter of iterating the list. We'll need several variables:

- A resulting map
- An index variable
- A variable for the current sum

Since it's possible to finish the algorithm without reaching the desired sum, we'll keep track of the current amount taken in a separate variable (when we're done, we'll check it against the desired sum to see if we got a solution or not).

```
Map<Integer, Integer> chosenCoins = new LinkedHashMap<>();
int currentSum = 0;
int coinIndex = 0;
```

Having these variables, when do we stop taking coins? There are two possibilities:

- We have reached the desired sum
- We ran out of coin values

We can put these two conditions in a while loop like this:

```
while (currentSum != targetSum && coinIndex < sortedCoins.size()){
    // TODO
}
```

Inside the body of the while loop, we need to decide how many coins to take of the **current value**. We take the current value from the list. We have its index:

```
int currentCoinValue = sortedCoins.get(coinIndex);
```

So far, we've accumulated some amount in the **`currentSum`** variable, the difference between **`targetSum`** and **`currentSum`** will give us the remaining sum we need to obtain:

```
int remainingSum = targetSum - currentSum;
```

So, how many coins do we take? Using integer division, we can just divide **`remainingSum`** over the current coin value to find out:

```
int numberOfCoinsToTake = remainingSum / currentCoinValue;
```

All we have to do now is put this information in the resulting dictionary as a key-value pair (only if we can take coins with this value), then increment the current index to move on to the next coin value:

```
if(numberOfCoinsToTake > 0){
    // TODO: add info to chosenCoins associative array (coin value => number of coins)
    // TODO: increase currentSum with total value of coins
}

coinIndex++;
```

Finally, return the resulting map.

4. Set Cover*

Write a program that finds **the smallest subset of `sets`,** which **contains all elements** from a given **sequence**.

In the Set Cover Problem, we are given two sets - a set of sets (we'll call it **`sets`**) and a **`universe (a sequence)`**. The **`sets`** contain all elements from **`universe`** and no others, however, some elements are repeated. The task is to **find the smallest subset of `sets` which contains all elements in `universe`.** Use the **Main** class from your skeleton.

Examples

Input	Output

Universe: 1, 2, 3, 4, 5 Number of sets: 4 1 2, 4 5 3	Sets to take (4): { 2, 4 } { 1 } { 5 } { 3 }
Universe: 1, 2, 3, 4, 5 Number of sets: 4 1, 2, 3, 4, 5 2, 3, 4, 5 5 3	Sets to take (1): { 1, 2, 3, 4, 5 }
Universe: 1, 3, 5, 7, 9, 11, 20, 30, 40 Number of sets: 6 20 1, 5, 20, 30 3, 7, 20, 30, 40 9, 30 11, 20, 30, 40 3, 7, 40	Sets to take (4): { 3, 7, 20, 30, 40 } { 1, 5, 20, 30 } { 9, 30 } { 11, 20, 30, 40 }

Greedy Approach

Using the greedy approach, at each step we'll take the set which contains the most elements present in the universe which we haven't yet taken. At the first step, we'll always take the set with the largest number of elements, but it gets a bit more complicated afterwards. To simplify our job (and not check against two sets at the same time), when taking a set, we can remove all elements in it from the universe. We can also remove the set from the sets we're considering.

Greedy Algorithm Implementation

You are given sample input in the **`main()`** method, your task is to complete the **`chooseSets()`** method:

```java
public static void main(String[] args) throws IOException {
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    String[] elements = reader.readLine().substring(10).split( regex: ", ");
    int[] universe = new int[elements.length];
    for (int i = 0; i < elements.length; i++) {
        universe[i] = Integer.parseInt(elements[i]);
    }
    int numberOfSets = Integer.parseInt(reader.readLine().substring(16));
    List<int[]> sets = new ArrayList<>();
    for (int i = 0; i < numberOfSets; i++) {
        String[] setElements = reader.readLine().split( regex: ", ");
        int[] set = new int[setElements.length];
        for (int j = 0; j < setElements.length; j++) {
            set[j] = Integer.parseInt(setElements[j]);
        }
        sets.add(set);
    }
    List<int[]> chosenSets = chooseSets(sets, universe);

    StringBuilder sb = new StringBuilder();
    sb.append(String.format("Sets to take (%d):%n", chosenSets.size()));
    for (int[] set : chosenSets) {
        sb.append("{ ");
        sb.append(Arrays.toString(set).replaceAll( regex: "\\[|]", replacement: ""));
        sb.append(" }").append(System.lineSeparator());
    }
    System.out.println(sb);
}

public static List<int[]> chooseSets(List<int[]> sets, int[] universe) {
    // TODO
}
```

The method will return a list of arrays, so first thing's first, initialize the resulting list:

```java
List<int[]> selectedSets = new ArrayList<>();
```

As discussed in the previous section, we'll be removing elements from the universe, so we'll be repeating the next steps until the universe is empty:

```java
while (!universeSet.isEmpty()) {
    // TODO
}

return selectedSets;
```

The hardest part is selecting a set. We need to get the set which has the most elements contained in the universe. We need to find the one with most elements in the universe:

```
int notChosenCount = 0;
int[] chosenSet = sets.get(0);
for (int[] set : sets) {
    int count = 0;
    for (int elem : set) {
        if (universeSet.contains(elem)) {
            count++;
        }
    }

    if (notChosenCount < count) {
        notChosenCount = count;
        chosenSet = set;
    }
}
```

The above code find the one **set** with most elements contained in the universe

Once we have the set we're looking for, the next steps are trivial. Complete the TODOs below:

```
// TODO: add chosenSet to result (selectedSets)
// TODO: remove all elements in chosenSet from universeSet
```

This is all, we just need to run the unit tests to make sure we didn't make a mistake along the way.

II. Simple Sorting Algorithms

5. Merge Sort*

Sort an array of elements using the famous merge sort.

Examples

Input	Output
5 4 3 2 1	1 2 3 4 5

Hints

Create your **`mergeSort`** method in **Main** class:

```
private static int[] mergeSort(int[] array) {

}
```

First if we have reached the bottom of the recursion (all elements have been split into partitions of 1), we return the whole array (1 element):

```
if(array.length == 1) {
    return array;
}
```

Extract the **index**, at the middle of the array, get the **lengths** and **initialize** the two partitions of the array:

```
int halfIndex = array.length / 2;

int firstArrayLength = halfIndex;
int secondArrayLength = array.length - halfIndex;

int[] firstPartition = new int[firstArrayLength];
int[] secondPartition = new int[secondArrayLength];
```

Fill the partitions with values from the main array:

```
for (int i = 0; i < firstArrayLength; i++) {
    firstPartition[i] = array[i];
}

for (int i = firstArrayLength; i < firstArrayLength + secondArrayLength; i++) {
    secondPartition[i - firstArrayLength] = array[i];
}
```

Recursively, do the same for each of the two partitions of the array. Each partitions gets split into two, until you reach partitions of one element:

```
firstPartition = mergeSort(firstPartition);
secondPartition = mergeSort(secondPartition);
```

From here on, it's the backtrack of the recursion. The main logic after the split of the partitions.

This is the main index, which will be used to follow the progress on the main array, and these are the indexes for the two partitions, which will be used to follow the progress on them:

```
int mainIndex = 0;

int firstPartitionIndex = 0;
int secondPartitionIndex = 0;
```

Here starts the main comparing algorithm. The loop's condition consists of both indexes, being compared to their corresponding partition lengths. Both partition's indexes will be increased, until one of the arrays is expired. In other words... This loop will go through both partitions, simultaneously, and will finish only when, one of the two indexes, reaches its partition's length.

```
while (firstPartitionIndex < firstArrayLength && secondPartitionIndex < secondArrayLength) {

}
```

Here is the comparison part. We **compare** the **current** element from the first partition, with the **current** element from the second partition.

In case the first partition's **current** element is **lower** by comparison, it will be put on the **current** position of the **main** array. If that is **NOT** the case, the second partition's **current** element, will be put on the **current** position of the **main** array. If you switch the comparing symbol, you might achieve a descending order in the sort. Currently the algorithm sorts in **ASCENDING** order.

```
while (firstPartitionIndex < firstArrayLength && secondPartitionIndex < secondArrayLength) {
    if (firstPartition[firstPartitionIndex] < secondPartition[secondPartitionIndex]) {
        array[mainIndex] = firstPartition[firstPartitionIndex];

        mainIndex++;
        firstPartitionIndex++;
    } else {
        array[mainIndex] = secondPartition[secondPartitionIndex];

        mainIndex++;
        secondPartitionIndex++;
    }
}
```

When the loop finishes, naturally, one of the two partitions, should be expired. In other words. One of the two partitions' values have been traversed totally. That would mean that, the other array would have some leftover values, which is why we need to store even them. Due to the fact we have nothing to compare them with, we just store them in the remaining positions of the main array:

```
while (firstPartitionIndex < firstArrayLength) {
    array[mainIndex] = firstPartition[firstPartitionIndex];

    mainIndex++;
    firstPartitionIndex++;
}

while (secondPartitionIndex < secondArrayLength) {
    array[mainIndex] = secondPartition[secondPartitionIndex];

    mainIndex++;
    secondPartitionIndex++;
}
```

At the end the other exit point of the recursive algorithm. Return the processed array:

```
return array;
```

6. Sorting*

Sort an array of elements using the famous quicksort.

Examples

Input	Output
5 4 3 2 1	1 2 3 4 5
1 4 2 -1 0	-1 0 1 2 4

Hints

You can learn about the Quicksort algorithm from Wikipedia. A great tool for visualizing the algorithm (along with many others) is available at Visualgo.net.

The algorithm in short:

- Quicksort takes unsorted partitions of an array and sorts them
- We choose the **pivot**
  - We pick the first element from the unsorted partition and move it in such a way, that all smaller elements are on its left and all greater, to its right
- With pivot moved to its correct place, we now have two unsorted partitions – one to the left of it and one to the right
- **Call the procedure recursively** for each partition
- The bottom of the recursion is when a partition has a size of 1, which is by definition sorted

First, define the **sorting method**:

```
private static void sort(int[] arr) {
    sort(arr, low: 0, high: arr.length - 1);
}

// arr[] --> Array to be sorted,
// low  --> Starting index,
// high  --> Ending index
private static void sort(int arr[], int low, int high) {

}
```

First, find the pivot index and rearange the elements, then sort the left and right partitions recursively. Now to choose the pivot point we need to create a method called **`partition()`**:

```
private static void sort(int arr[], int low, int high) {
    if (low < high) {
        /* pi is partitioning index, arr[pi] is
          now at right place */
        int pi = partition(arr, low, high);

        // Recursively sort elements before
        // partition and after partition
        sort(arr, low, high: pi - 1);
        sort(arr, low: pi + 1, high);
    }
}
```

```
/* This method takes last element as pivot,
   places the pivot element at its correct
   position in sorted array, and places all
   smaller (smaller than pivot) to left of
   pivot and all greater elements to right
   of pivot */
private static int partition(int arr[], int low, int high) {

}
```

```
int pivot = arr[high];
int i = (low - 1); // index of smaller element
for (int j = low; j < high; j++) {
    // If current element is smaller than or
    // equal to pivot
    if (arr[j] <= pivot) {
        i++;

        // swap arr[i] and arr[j]
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}

// swap arr[i+1] and arr[high] (or pivot)
int temp = arr[i + 1];
arr[i + 1] = arr[high];
arr[high] = temp;

return i + 1;
```

III. Searching Algorothms

7. Searching*

Implement an algorithm that finds the index of an element in a sorted array of integers in logarithmic time

Examples

Input	Output	Comments
1 2 3 4 5 1	0	Index of 1 is 0
-1 0 1 2 4 1	2	Index of 1 is 2

Hints

First, if you're not familiar with the concept, read about binary search in Wikipedia. Here you can find a tool which shows visually how the search is performed.

In short, if we have a **sorted collection** of comparable elements, instead of doing linear search (which takes linear time), we can eliminate half the elements at each step and finish in logarithmic time. Binary search is a **divide-and-conquer** algorithm; we start at the middle of the collection, if we haven't found the element there, there are three possibilities:

- The element we're looking for is smaller – then look to the left of the current element, we know all elements to the right are larger
- The element we're looking for is larger – look to the right of the current element
- The element is not present, traditionally, return -1 in that case

Start by defining a class with a method:

```
public static int getIndex(int[] arr, int key) {

}
```

Inside the method, define two variables defining the bounds to be searched and a while loop:

```
public static int getIndex(int[] arr, int key) {
    int start = 0;
    int end = arr.length - 1;

    while (start <= end) {
        // TODO: Find index of key
    }

    return -1;
}
```

Inside the while loop, we need to find the midpoint:

```
int mid = start + (end - start) / 2;
```

If the key is to the left of the midpoint, move the right bound. If the key is to the right of the midpoint, move the left bound:

```
if (key < arr[mid]) {
    end = mid - 1;
} else if (key > arr[mid]) {
    start = mid + 1;
} else {
    return mid;
}
```