Assessment 2 – Recursive solutions

In this assessment, you will be tasked with implementation of several recursive solutions for two different computational problems.

How to submit

All of your answers should be submitted in one file. The file should be named Assessment2.java and contain one java class called Assessment2. Each task requires you to write one or more java methods. Write those as public static methods of the Assessment2 class. Please also add your name to the @author tag on top of the file. The submission link will appear on moodle in the A2 - Programming Assessment section.

An example submission file will be provided for you in the same section. It will contain the Assessment2 class with some empty methods written for you, so you can use it as a starting point for you implementations. Marking of this assessment will include some automated tests, so please make sure your Assessment2.java file compiles correctly and your implementations can be accessed by calling the methods with the same signatures as the methods that appear in the example submission file.

Additionally, the Assessment2.java file provided on moodle will contain a main method with some simple pre-written tests. You can run the main method at any time to check if your methods can be correctly accessed by tests and if they can pass simple tests. The main method makes calls to the exact methods that will be tested during automated marking, so if running main method causes some errors, it will likely cause the same errors during automated marking.

Marking

There are a total of 100 marks in this assessment. Each task will be marked separately using automated tests. Tests will consist of a number of test cases, including some edge cases, such as trying to mergesort an array of size 1 or trying to find the maximum in a 1x1 mine, so make sure to account for those. There will be no tests with null values, so you can safely assume parameters will never be nulls. Any solutions that do not compile will get a mark of 0 for each task that does not compile, i.e. a compilation error in the mergesort method would result in a mark of 0 for task 1.2 but will not affect marks for other tasks.

Use of java libraries is generally prohibited and will result in deductions of marks, please write your own implementations. The exceptions are all methods from java.util.Arrays that are not related to sorting, and all methods from java.lang.Math.

Any cases of suspected plagiarism will be investigated in line with University Policies and Procedures regarding Academic Misconduct https://www.kent.ac.uk/ai/academicpolicies.html.

Questions and Help

If you are struggling to start the assessment, you can start by reviewing the Mergesort lecture from week 17. It includes a full description of the Mergesort algorithm and some useful psedocode for tasks 1.1 and 1.2.

If you have any questions about the assessment, you can email Sergey at so362@kent.ac.uk .

Task 1. Mergesort [60 Marks total]

This task is about implementing the Mergesort algorithm with a few variations.

Task 1.1 Merge [10 Marks]

Merge is a method used to merge two sorted arrays into one sorted array. It takes two integer arrays as inputs and returns an integer array.

Implement the merge method.

Task 1.2 Mergesort [10 Marks]

Mergesort is a method that takes an integer array and returns a sorted copy of the same array. It first sorts the left half of the input array, then the right half of the input array, and finally merges the two sorted halves together into a sorted version of the input array.

Implement the mergesort method.

Task 1.3 Merge3 [10 Marks]

In the Mergesort lecture in week 17 we have proven that the computational time complexity of mergesort is $O(n * \log_2 n)$. Perhaps the time complexity of mergesort can be improved further by separating the array into three parts instead of two.

Start by implementing a method called merge3. This method should take three sorted arrays as inputs and return a sorted array consisting of all elements of the three input arrays.

Task 1.4 Mergesort3 [10 Marks]

After completing task 1.3, you should have a working merge3 method that allows you to merge three sorted arrays at the same time.

Use this method to implement Mergesort3 method that sorts the input array by dividing it into three parts, sorting those parts recursively using recursive calls to itself, and using the merge3 method to merge them into a sorted array.

Task 1.5 MergeAll [10 Marks]

Now that we have written merge methods for merging two and three arrays at the same time, let's write a merge method that can take any number of sorted arrays and merge them all into one sorted array. This method will take an array of integer arrays as input and return one integer array as output.

The implementation of this method should not use calls to any other methods.

Task 1.6 MergesortK [10 Marks]

Let's finish our exploration of mergesorts by implementing a parametrized version of mergesort that will allow us to choose how many parts the input array will be divided into.

This method will take an integer array and integer K as inputs and produce one integer array as output. It will sort the input array by dividing it into K parts, recursively sorting those parts, and using the mergeAll method to merge the K sorted arrays into one sorted array.

Implement the mergesortK method.

Task 2. Digger [40 Marks total]

This scenario will be familiar to most of you from Assignment 4 of the COMP3830 module from last year. Here is the scenario of the problem:

You have been hired at an open-air mine. You are to write a program to control a digger. For your task, you have been given a 'map' of all the underground resources in the mine. This map comes as a two-dimensional integer array. The array has n rows and k columns. Each integer is between zero and one hundred (inclusive). This array represents a grid map of the mine – with each row representing a different height level. Each square in the grid can contain valuable resources. The value of these resources can go from zero (no resources) to 100 (max resources).

The grid maps the resources just below the surface, and down to the lowest diggable depths. The digger starts at the surface (so, just on top of the topmost row in the grid)—at any horizontal position between 1 and k. The digger cannot dig directly downwards, but it can dig diagonally in either direction, left-down, or right-down. In its first time-step, it will dig onto the first row of the grid. In its second time-step, it'll hit the second row, and so on. When the digger reaches row number n, it has to be air-lifted out of the mine, and is no longer usable. Every time the digger hits a square with resources in it, it collects those resources.

We want to find out the maximum number of resources a digger can collect from this mine in one run. Let's implement a recursive solution to this problem, reducing the problem of finding the maximum number of resources reachable from any point in the grid to finding the maximum number of resources that can be achieved by moving down-left, then finding the maximum number of resources by moving down-right, and selecting the highest of the two.

Task 2.1 Recursive maxResources [20 Marks]

Let's write a method called maxResources. This method will take a two-dimensional integer array as input and produce one integer output. The input represents the grid map of the mine and the integer output should be equal to the maximum amount of resources that can be collected by a digger in a single run.

We want to have a method that is implemented recursively, so it may be useful to implement a version with additional inputs to allow it to keep track of which cell it is looking at.

The recursive formula should define maxResources for a particular coordinate (x,y) as the sum of the number of resources in (x,y) and the value of maxResources from either coordinate (x+1, y+1) or coordinate (x-1, y+1), whichever is higher.

Task 2.2 maxResourcesM [20 Marks]

If you have implemented the maxResources method and tried testing it with large input sizes, you may have noticed that it starts to run very slowly after a certain depth. This is because the time complexity of the maxResources method is very large, close to $O(2^n)$. Let's improve this method by adding memoization to its implementation.

Make a copy of the original maxResources method and call it maxResourcesM. Re-implement this method in a way that would allow it to store the results of previous calls and use those when the same calls are made again.