ECS510U Algorithms and Data Structures in an Object-Oriented Framework

Exercise Sheet 6: Developing an Algorithm to Solve a Problem

This is an exercise to support the "Sorting and Efficiency" section

Algorithms

This exercise is intended to get you to think in terms of algorithms. You need to think first about an algorithm to solve this problem. Once you have thought of an algorithm, then you can write a Java static method to implement it. You could then see if you could think of a better algorithm.

Although this is not a problem that involves sorting, like sorting it is one which can be tackled with a variety of algorithms. Also as with sorting algorithms, this will also test and develop your general skills at programming with indexed collections. The problem here is a well-understood one, but I have deliberately not given you the name by which it is known because I want you to work out your own algorithm to solve it. If you knew its name and you decided to use Google to find an algorithm for it, rather than work it out yourself, it would spoil the point of this exercise.

You should approach the problem by first developing an algorithm which works, without being concerned about efficiency. Once you have done this, you should experiment with variations or different algorithms to see how that changes the time taken to solve the problem or the size of the problem it can deal with in a reasonable amount of time. Careful thought may lead you to being able to think of a more efficient algorithm than what you first came up with.

Here is the problem:

Given a list of integers, A_0, A_1, A_2, ... , A_{n-1}, each of which may be positive or negative, find the sublist of integers A_p, ..., A_q which has the maximum sum of any sublist, and return the sum. If all the integers are negative, return 0.

A sublist has to start at some position in the original list, finish at some later position and include every number which is in the original list between those positions. So, for example if the list is `{10,-20,11,-4,13,3,-5,-17,2,15,1,-7,8}` then the answer is `23` since this is the sum of the list `{11,-4,13,3}` and no other sublist has a higher sum. The answer is always at least `0`, because the empty list is always a possible sublist, and its sum is `0`.

Assume the numbers are stored in a Java array `a` so `a[k]` stores A_k for any `k` between and including `0` and `n-1`. Then write a Java method which returns the sum of some portion of the array where no other portion has a higher sum.

One possible use of this algorithm is in stock market analysis. Suppose the integers represent the rise or fall of a share price over a period of time. Then the best period to own that share is the period represented by the solution to this problem.

A program which provides a frame to test the method can be found in the code folder for this section in the files `Exercise6.java` and `Exercise6a.java`. You will find in these files that only the header for the method `highestSum` is given. This method is intended to take an array storing some integers as its parameter and return the sum of that portion of the array which has a higher sum of any other portion. You have to write the code for the method which makes it do this. So, the programs you are given provide you with the support code so you can concentrate on the important issue, developing and coding the algorithm.

Continued overleaf

In the file `Exercise6.java` you type in your own numbers for the list, but in `Exercise6a.java`, a list of numbers is generated at random for you.

In `Exercise6a.java`, you are asked to enter a "seed", the length of the list, and the highest integer value, *max*. Integers are generated at random, both positive and negative from the range *-max* to *max*. The program takes the time before and after the method that gives the algorithm is executed. This gives a rough estimate of the timing (not an exact figure, because typically your computer will spend some of its time doing other things, e.g. maintaining the clock display, when it's obeying your code).

The "seed" is just a way of ensuring that when you run your code, the particular list of numbers associated with the seed will be generated, so if you change your code and run it again, you can make sure it's using the same "random" numbers as it did before. Apart from that, the numbers will not appear to have any pattern.

For an example solution, if you type 1234 for the seed, 500 for the length of the list, and 100 for the highest integer, the answer you should get is 1212 (which is the sum of the range from position 316 to 476 inclusive). To keep things simple, in the method that is the answer to this exercise, as well as returning the maximum sum of a sublist, your code should also print the starting and finishing positions of the sublist that has the maximum sum.

Efficiency and Correctness

With this problem, a very poor algorithm will take a long time (time for you to sit and wait for it to terminate) to find the sublist with the highest sum of even 1000 integers. The algorithm you are most likely to come up with after some thought on efficiency will work fine with a list of 1000, but once the size of the array gets to around 100000, you will see a delay between starting it and getting the solution. The best algorithm, however, will take a short time, appearing to be instantaneous, for 100000 integers, and only when the length of the list gets into the millions will there be a delay long enough to notice in giving a solution.

In the module notes on efficiency, you will see how algorithms can be categorised using the "big-O" notation. Try and work out the big-O category your algorithm falls into.

It is, of course, important that a program works correctly. With an algorithm, this is not a matter of trial and error, you should be able to argue why the algorithm and your code which implements it gives a correct solution. As you have seen in previous exercises, it is easy to write code that you think works, but would not work in other cases that you did not think of to test it. So your explanation should be able to cover why you think it will work in all cases.

As this exercise is all about developing the algorithm yourself, the assessment will require you to be able to explain it and why you think it works. Code which works but you cannot explain how and why will not get high marks.

Matthew Huntbach