COMP202 **Programming Assignment**
Complexity of Algorithms **21 March 2023**

Title: Square Grouping

Submissions should be made via CANVAS by deadline 26 April, 16:00:
`https://liverpool.instructure.com/`

Go to COMP202 and to Assignment:
COMP202: CA2 – PROGRAMMING ASSIGNMENT 2023

Notes:

1. This assessment is worth 15% of your overall course grade.
2. Standard late penalties apply, as per university policy.
3. Learning outcomes covered by this CA task will also be covered within resit exam: this prevents the need for explicit reassessment of this component. The resit exam will replace the CA component in case this is failed.

Learning outcomes

The purpose of this exercise is for you to demonstrate the following learning outcomes and for me to assess your achievement of them.

1. To demonstrate how the study of algorithmics has been applied in a number of different domains.
2. To introduce formal concepts of measures of complexity and algorithms analysis.
3. To introduce fundamental methods in data structures and algorithms design.

Note: You will be provided with a collection of sample inputs together with correct answers to these inputs. You should aim at submitting your final program only if it produces correct answers to all these inputs.

Academic integrity

The work that you submit should be the product of yourself (alone), and not that with any other student or outside help. Obviously I am providing a source code framework within which you will provide your own method for solving this problem, but the code that you write within this framework should be your own code, not that obtained in collaboration with other students or other outside assistance or sources.

Problem Description

The Square Grouping problem is defined as follows. You are given as input a sequence of n *positive* integers $a_1, a_2, \ldots, a_n$ and a *positive* integer $k \leq n$. Your task is to group these numbers into k groups of *consecutive* numbers, that is, as they appear in the input order $a_1, a_2, \ldots, a_n$, so as to minimise the total sum of the squares of sums of these numbers within each group.

More formally, you have to decide the $k-1$ *cutting* positions $1 \leq i_1 < i_2 < \cdots < i_{k-1} \leq n-1$, where (we assume below that $i_0 = 1$):

- group G_1 is defined as $G_1 = \{a_1, a_2, \cdots, a_{i_1}\}$
- group G_j, for $j = 2, 3, \ldots, k-1$, is defined as

$$G_j = \{a_{i_{j-1}+1}, a_{i_{j-1}+2}, \cdots, a_{i_j}\}$$

- group G_k is defined as $G_k = \{a_{i_{k-1}+1}, a_{i_{k-1}+2}, \cdots, a_{i_k}\}$

Then, such feasible solution, that is, grouping into these k groups, has the value of the *objective function* equal to

$$\sum_{j=1}^{k} \left(\sum_{a_i \in G_j} a_i \right)^2 .$$

Your goal is to find such grouping into k groups so that this objective function is minimised.

Suppose, for instance, that $n = 5$ and $k = 3$ and that input sequence is:

$$5\ 7\ 11\ 4\ 21,$$

that is, $a_1 = 5, a_2 = 7, a_3 = 11, a_4 = 4, a_5 = 21$. Then, for example, setting the $k - 1 = 2$ cutting positions as follows

$$5\ 7\,|\,11\,|\,4\ 21,$$

that is the cutting positions $i_1 = 2, i_2 = 3$, define the following $k = 3$ groups $G_1 = \{5, 7\}$, $G_2 = \{11\}$, $G_3 = \{4, 21\}$. The objective function value of this grouping is

$$(5+7)^2 + (11)^2 + (4+21)^2 = 144 + 121 + 625 = 890.$$

Observe that there is a better solution here, with the following grouping

$$5\ 7\,|\,11\ 4\,|\,21,$$

The objective function value of this grouping is

$$(5+7)^2 + (11+4)^2 + (21)^2 = 144 + 225 + 441 = 810,$$

and it can be checked that this is the optimal grouping, that is, 810 is the smallest possible value of the objective function among all possible groupings in this instance.

For further examples of inputs together with values of their optimal solutions, see the text file data2.txt that I provide (see explanation of the data format below). In fact the example sequence above, 5 7 11 4 21, with $k = 3$ is the first instance in data1.txt and in data2.txt (data2.txt contains also solutions).

Observe that the input sequence $a_1, \ldots, a_n$ need not be sorted and you are not supposed to change the order of these numbers, but only find appropriate $k - 1$ cut points that define the k groups (each group must be non-empty, that is, must contain at least one number). The task is, given *any* sequence of n (strictly) positive integers and $k \leq n$, find the grouping that has the smallest possible objective function value. Note that the input sequence may contain the same number multiple times.

Also observe that it is possible that $k = n$ in the input to this problem (it is impossible that $k > n$, though). For instance if the input sequence is as above

$$5\ 7\ 11\ 4\ 21,$$

and $k = n = 5$, then there exists only one possible feasible grouping into $k = 5$ groups with the following cut points:

$$5\,|\,7\,|\,11\,|\,4\,|\,21,$$

and the objective value of this grouping is

$$(5)^2 + (7)^2 + (11)^2 + (4)^2 + (21)^2 = 25 + 49 + 121 + 16 + 441 = 652,$$

and this is the (optimal) solution to this instance with $k = 5$.

You should write a procedure that for any given input sequence of n positive integers and any given $k \leq n$, finds a grouping with minimum value of the objective function value. Your procedure should only output the value of the objective of this optimal solution (grouping). That is, it should compute the grouping with minimum possible value of the objective function among all feasible groupings, and then output the objective value of this optimal grouping.

Additionally, you should include a brief idea of your solution in the *commented* text in your code and you should also include a short analysis and justification of the running time of your procedure. These descriptions are part of the assessment of your solution.

Hints

Observe that in this problem we have two kinds of objects to maintain: the numbers $a_1, \ldots, a_n$ and groups $G_1, \ldots, G_k$, which suggests that the dynamic

programming solution should involve two kinds of indices. Similarly to this, when dealing with dynamic programming solution to the $\{0,1\}$ Knapsack problem, we also used two kinds of indices, see lecture notes on Fundamental methods – Dynamic Programming. I then suggest you to start your investigations with making some simple observations similar to those we have made in these lecture notes for the $\{0,1\}$ Knapsack and Weighted Interval Scheduling problems, respectively, namely that the optimal solutions to these problems either contain the last item (interval, resp.) n or not. Namely, ask yourself the question: what if you already knew the cutting position of the last group? And, as usual, you should start by defining appropriate dynamic programming table and come up with the appropriate recurrence solution to the problem. After you have your recursive solution, you should translate it to a sequential solution that fills in the table in an appropriate way.

Programming Language

You will be using Java as the programming language.

Program framework description

IMPORTANT: Before submitting, you must rename your file `Main123456789.java` where `123456789` is replaced with all digits of your Student ID. You also must rename the main public class `Main123456789{ }` in your file by also replacing `123456789` by all digits of your Student ID.

I provide a template program called `Main123456789.java` that you will use (without altering anything but the place to put your code) to include the code of your procedure to solve the Square Grouping problem.

To use this template, after you write your code inside of method called SquareGrouping, you must include in the *current directory* the input text files data1.txt and data2.txt. Note, however, that if you need any additional procedures, you may include them outside of the text of the procedure SquareGrouping.

To compile your program in command line (under Linux/Unix) use something like this (this may differ within your system):

```
javac Main123456789.java
```

Then, you can run your program from command line like this

```
java Main123456789 -opt1
```

which will run the program with data1.txt as input file and will output answers to all instances in order they appear in data1.txt. You may use your own data1.txt in the format (see below) to experiment with your program. Input file data1.txt may contain any number of correct input sequences.

Now, if you run the program with

```
java Main123456789 -opt2
```

this will run the program with data2.txt as input file. In this case, the output will be the indices of inputs from data2.txt that were solved incorrectly by your program together with percentage of correctly solved instances. If all instances are solved correctly, the output should be 100%. File data2.txt contains the same instances as data2.txt, but, in addition data2.txt also contains the correct answers to these instances. You may use data2.txt to test the correctness of your program.
Description of the input data format:

Input data text file data1.txt has the following format (this example has 2 inputs, each input ends with A):

k
a_1
a_2
...
...
a_n
A
k
a_1
a_2
...
...
a_n
A

In general file data1.txt can have an arbitrary number of distinct inputs of arbitrary, varying lengths. File data1.txt contains 34 different instances of the problem. Observe that n is not explicitly given as part of the instance.

Input data text file data2.txt has the following format (this example has

again 2 inputs, each input ends with A):

$$
\begin{array}{c}
k \\
ans_1 \\
a_1 \\
a_2 \\
... \\
... \\
a_n \\
A \\
k \\
ans_2 \\
a_1 \\
a_2 \\
... \\
... \\
a_n \\
A
\end{array}
$$

There ans_1 is the correct value of the optimal (minimum) value of the objective function in the optimal solution to the first and second instance, respectively. File data2.txt contains the same 34 instances of the problem as in file data1.txt but in addition has answers. This data can be used to test the correctness of your procedure.

Again, in general file data2.txt can have an arbitrary number of distinct input sequences of arbitrary, varying lengths. It is provided by me with correct answers to these instances.

The solutions shown in data2.txt are (at least) the claimed solutions for each sample input, computed by my program. Recall that your solution should print out the objective function value of the grouping in the given sequence that is the grouping with the smallest possible objective function value for the given instance. Note, that your program does not need to output the grouping itself (that is, the cutting positions), but only the objective function value of the optimal grouping.

Program submission

You must submit a **single** Java source code in a single file that must be called `Main123456789.java` (not the byte code), where `123456789` is replaced with all digits of your Student ID, via CANVAS:
`https://https://liverpool.instructure.com/`

Go to COMP202 and to Assignment: COMP202: CA2 – PROGRAMMING ASSIGNMENT 2023

IMPORTANT: Before submitting, you must rename your file `Main123456789.java` where `123456789` is replaced with all digits of your Student ID. You also must rename the main public class `Main123456789{ }` in your file by also replacing `123456789` by all digits of your Student ID.

Your source file `Main123456789.java` must have the (unaltered) text of the template provided by me, containing the text of your procedure inside the SquareGrouping method. You are allowed to include additional procedures outside the SquareGrouping method if needed. In addition, within commented parts of method SquareGrouping, you should describe your recursive solution and how you implement it sequentially. Moreover, you should also describe a short running time analysis of your sequential implementation in terms of n and big-O notation.

You are responsible that your source code program `Main123456789.java` can be compiled correctly with Java compiler and executed on (any) one of the computers in the Computer Science Department's that runs Java installation under Linux, where I will test your programs. You may also remotely connect to any Linux computer in the Department to compile/test your program. Programs that will not correctly compile on Departmental Linux Java installation will automatically receive mark 0 for the correctness part.

Assessment

Marking on this assessment will be performed on the following basis:

- Accuracy of solution (e.g., does it give correct answers for all of the test cases, and is it a general solution method for instances of this problem?): 60%
- Clarity of solution (Is your program easy to follow? Is it commented to help me see your solution method?): 10%
- Correctness of time complexity (in big-O notation of the problem size n) description of your procedure and description of your solution. (Have you provided an analysis of the (asymptotic) running time of your method, and is that analysis correct? Is the description of your solution (recursion and sequential implementation) correct and clearly written?: 20%
- Optimality of solution (Do you have the "best", i.e., quickest, solution possible in terms of the asymptotic runtime?): 10%