Practical assignment 2
Algorithms & Datastructures

Carlo Jessurun s1013793
Tony Lopar s1013792

Nijmegen, January 11, 2018

Frits Vaandrager

Joshua Moerman

2017-2018
Radboud University Nijmegen

Contents

1 Explanation

In this chapter we will explain how our algorithm works. The explentation is split into three parts.

1.1 Reading the input

For the reading of the input we made a custom reader which replaces the scanner in Java. The custom reader works faster than the standard scanner in Java. At first we reused our last implementation of the scanner class we did in the first assignment. In this instance however, we found it to perform quite slowly and it was definitely holding the speed back a lot. So after researching replacements for the scanner class we found that the BufferedReader was a way faster class to use. To give a short summary of our research we found the following information about both the Scanner and the BufferedReader:

The java.util.Scanner class is a simple text scanner which can parse primitive types and strings. It internally uses regular expressions to read different types.

Java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of sequence of characters.

- BufferedReader is synchronous while Scanner is not. BufferedReader should be used if we are working with multiple threads.
- BufferedReader has significantly larger buffer memory than Scanner.
- The Scanner has a little buffer (1KB char buffer) as opposed to the BufferedReader (8KB byte buffer), but it's more than enough.
- BufferedReader is a bit faster as compared to scanner because scanner does parsing of input data and BufferedReader simply reads sequence of characters.

Since we found that is fast, it's still not recommended as it requires lot of typing. The BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. With this method we will have to parse the value every time for desired type.

At this point we decided to use our own implementation of a Reader class with the use of DataInputStream. This provided us with a really fast way to read the input given by this assignment. We found it to be the fastest of all our tested reader implementations but it does require very cumbersome methods in its implementation. It uses DataInputStream to read through the stream of data and uses read() method and nextInt() methods for taking inputs. This is by far the fastest ways of taking input but is difficult to remember and is cumbersome in its approach. Our implementation can

be found in the "Reader.java" class. We used the test data we got from the algorithms website but also tested our other scanners against the website: "http://www.spoj.com/problems/INTEST/" which is basically a huge place to test how fast you can read data. The "Reader.java" class also performed really fast on this testing framework.

Since we did not want to constantly depend on testing with the ads.cs.ru.nl framework we copied the tests to a local machine. We've used JUnit to make unittests and used those to test, debug and validate our code. The implementation (not necessary) can be seen in "ExampleCases.java". Those 45 testcases include those of the testframework in ads.cs.ru.nl and we saw a very significant improvement on the runtime after implementing the new Reader class. In the end, comparing with the default scanner implementation we saved a lot of time and made reading input literally ten times faster over the entire testset.

- **Scanner implementation:** 400ms
- **DataInputStream implementation:** 38ms

1.2 Algorithm

The algorithm first tries to make profitable groups of the input. Products are added to the previous group when after addition the group still gives profit.

After this the algorithm optimizes the list with profits by putting groups with a higher profit in front of the list.

Finally the first D profits are substracted from the total sum of products where D is the number of dividers.

1.3 Computing output

2 Analysis

2.1 Correctness

In this section we will discuss the Correctness of the processes.

2.1.1 Reading the input

2.1.2 Algorithm

2.1.3 Computing output

2.2 Complexity

In this chapter we will describe the complexity of our algorithm. We will first describe the complexity in detail for the smaller parts and after this compute the complexity for the whole algorithm.

2.2.1 Reading the input

The reading of the output has to process all items in the output for the algorithm. Before we read the prices of the products, we have to read the number of products and the number of dividers. This means that in total $n + 2$ items should be read from the input. The complexity of this reading is $\mathcal{O}(n)$.

2.2.2 Algorithm

2.2.3 Writing output

The output gets the total sum of products with the possible profit already substracted. The output should only be rounded, since the substraction may cause values that does not end with 0. The complexity of this process is $\mathcal{O}(1)$.

3 Reference