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 **3**
1.1 Reading the input 3
1.2 Algorithm 4
1.3 Computing output 4

2 Analysis **5**
2.1 Correctness 5
2.1.1 Reading the input 5
2.1.2 Algorithm 5
2.1.3 Computing output 5
2.2 Complexity 5
2.2.1 Reading the input 5
2.2.2 Algorithm 5
2.2.3 Writing output 6

3 Reference **7**

1 Explanation

In this chapter we will explain how our algorithm works. The explanation 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”[1] 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[2] 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. Profitable groups would consist of numbers [1..4]. So for instance we might find the number 14 first after which we add 4 to a list to keep track of potential profit. However say we would encounter a list [1,1,1,1], the algorithm would then keep on counting 1’s until we get to 4 and then add four the the potential profit group.

After doing so, the next step includes analyzing which groups are the most profitable to based on the list we made earlier. 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. If we run out of dividers to place before we are doing with going through the products we terminate early and return the total optimized sum.

1.3 Computing output

This is simply the result of our algorithm. We have a “private static int sum” in our implementation which we constantly update with new prices and eventually as a last step will be returned when the algorithm is done.

[1]http://www.spoj.com/problems/INTEST/
[2]http://junit.org/junit4/

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. The parts which depend on the number of dividers are indicated by $|D|$.

- Reading the input: $\mathcal{O}(n)$
- Algorithm: $\mathcal{O}(n)$
- Writing output: $\mathcal{O}(1)$
- **Total algorithm:** $\mathcal{O}(n)$

2.2.1 Reading the input

The reading of the input has to process all items in the input 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 $\mathcal{O}(n)$.

The algorithm starts with reading the input. This is done in a loop which reads all the values from the input. The input starts with the number of products and dividers followed by the prizes of all products. This gives a complexity of $\mathcal{O}(n)$. The custom reader only opens and closes a connection which has a complexity of $\mathcal{O}(n)$, so the total complexity of reading the input is $\mathcal{O}(n)$.

2.2.2 Algorithm

The algorithm itself has 3 steps in total. The first step is to find groups with potential profit. Therefore, there is looped once over all items of the input. In the loop there is an if else which adds a group to the potentialProfitList or not. Whether we add a product to the list is done based on whether a given product itself is profitable or not. We calculate this by taking the mod 10 of a product's price and look if it is 4. If a combination of two, tree or four products together make 4 ([1,1,1,1]) we add those to the list as 4 as well. If we otherwise for example find a group of 3 after which the next value is 2, we then add the 3 and start with 2 as our new value to see if we can make a higher potential profit.

The complexity of this whole process depends on the number of executions in the loop. This gives a complexity of $\mathcal{O}(n)$.

The second part of the algorithm is to optimize the list of profitable groups. In this process a loop is executed which iterates over all groups found in the previous step. In the worst case the previous step only created groups with one product. This means that this process has a complexity of $\mathcal{O}(n)$.

The last step of the algorithm is substracting the first $|D|$ elements from the optimized list with profits. This process has a loop which iterates through all items of the optimizedProfitList. However, this loop is terminated when there are no dividers anymore. This means that the number of times the loop is executed is $|D|$ when $|D| \leq n$. So, the worst case is when there are more dividers than products. This results in a complexity of $\mathcal{O}(n)$ for this process.

The algorithm has three processes which are performed after each other. All of these processes have a complexity of $\mathcal{O}(n)$ which give a total complexity of $\mathcal{O}(3n)$. Since this complexity depends on n we may write it as $\mathcal{O}(n)$.

2.2.3 Writing output

The result of the the last process of the algorithm is given to the output. Before writing the output, this result is rounded up or down depending on the last digit. If the last digit is 0 until 4, the total is round down. Otherwise the total is rounded up. After this the rounded result is printed to the system. Both operations are performed in constant time which results in a complexity of $\mathcal{O}(1)$.

3 Reference

Sphere Research Labs, (2017). Input/Output performance testing framework.
http://www.spoj.com/problems/INTEST/

JUnit, (2017). JUnit is a simple framework to write repeatable tests.
http://junit.org/junit4/