

**MODULAR PROGRAMME**

**COURSEWORK ASSESSMENT SPECIFICATION**

**Module Details**

|  |  |  |
| --- | --- | --- |
| **Module Code** UFCFT4-15-3 | **Run** 18SEP/1 | **Module Title** CRYPTOGRAPHY |
| **Module Leader** Rong Yang | **Module Coordinator** Essam Ghadafi | **Module Tutors** Rong Yang |
| **Component and Element Number** B: CW1 | | **Weighting: (% of the Module's assessment)** 75% |
| **Element Description** PRACTICAL COURSEWORK (Practical Coursework) | | **Total Assignment time** 24 hours |

**Dates**

|  |  |
| --- | --- |
| **Date Issued to Students**  25th of Oct, 2018 | **Date to be Returned to Students**  4 working weeks after the submission |
| **Submission Place**  **Blackboard** | **Submission Date** 13/12/2018 |
| **Submission Time** **2.00 pm** |

**Deliverables**

|  |
| --- |
| See attached |

**Module Leader Signature**

|  |
| --- |
| Rong Yang |

Cryptography Programming Tasks

# Introduction

As part of the Cryptography module at University of the West of England I was required to complete 4 programming tasks of varying difficulty. The four tasks were;

1. Credit card validation.
2. BCH(10,6) generation, checking, and error correction.
3. Brute force password breaking of up to 6 characters.
4. Fermat and Dixon factorisation of numbers up to 10^10 and larger.

# What I’ve learned

I previously had negligible experience with java and having created java based NetBeans User Interfaces for each task I can now say I have a solid foundational understanding of creating java interfaces. However the amount of work completed in the background by the NetBeans IDE means that I would be unable to create a project from scratch without relying on the IDEs functionality.

Several times when I’ve been writing reports on a projects I have worked on it comes to what areas I would have like to put more work into it is difficult to come up with any ideas because the topic has been covered by the required task. Each task in this module has been different since instead it feels like each task focused on a specific aspect of the area to aid the learning.

If I didn’t have to meet the spec I would have delved deep into each task. The best example of this being the password breaking program. The current implementation has a drop down list with different methods. (Here I had wanted to use radioButtons and radioGroups however I had some difficulty implementing them correctly.)

This is where I was being overly ambitious in my plans to implement each password breaking method we were taught in the lectures. However while only brute force is currently implemented it could make for a good project to take on in my own time to improve my programming skills.

The credit card and BCH tasks helped in my understanding that a lot of thought is put into numbers that are used within our society. And the examples of valid credit card numbers and BCH checking are only simple examples.

The simple password breaking function I implemented made it really hit home just how long an exponential increase in time taken really is. Of course I theoretically understood how much of an increase it was but sitting there waiting for a 6 character long password to be broken made it that much clearer.

The factorisation of large integers was interesting to implement. I understood the pen and paper methods as taught in the tutorial but transfering that to a programming language was difficult in some places. The algorithm provided for Fermat factorisation made its implementation fairly easy. The difficulty arose with several parts of Dixons factorisation.

### The function I created was returning 1 \* n as powers of n, while technically correct this is not the desired output and I was unable to find the exact cause for this. As a work around I simple reject the powerSet that creates these powers and continue looking for an even powerSet. I am unsure whether this is the correct solution but it solves the problem. However the implementation of this has caused some redundancy in the code of task 4 that I have not had time to clean up before the deadline.

There were other Issues with 7-smooth, merging powerSets and selecting a reasonable size for the base.

Efficiency

Timings were recorded for tasks 3 and 4. This section will look at and analyse these results.

### Task 3

Since task 3 involved passwords and since passwords are likely to use natural language I hypothesised that changing the order of the character set (a-z 0-9) may decrease the time required to break passwords using natural language. So I ran all tests of the password breaking twice, once with character set (a-x 0-9), and again with a character set based on which character are used most in english language.

The timing results are displayed in milliseconds in the following table.

|  |  |  |
| --- | --- | --- |
| Password | Timing of character set (a-x 0-9) | Timing of character set (most common) |
| this | 1015 | 455 |
| is | 18 | 17 |
| very | 1165 | 1121 |
| simple | 978140 | 412590 |
| fail7 | 8454 | 24458 |
| 5you5 | 45066 | 42569 |
| 3crack | 1462480 | 1391144 |
| 1you1 | 38092 | 48173 |
| 00if00 | 1314119 | 1799917 |
| cannot | 144186 | 494395 |
| 4this4 | 1657574 | 1637589 |
| 6will | 48527 | 45078 |
| Total Timing = | 5698836 | 5897506 |

The table shows that as expected increasing the number of characters increases the time take to break it with brute force exponentially.

The time taken to break some passwords was decreased with a character set ordered by most common whereas the time was increased with others. Also the total time taken to break all passwords together was increased with (most common). This is most likely because the range of tests was not wide enough and did not properly compare natural language and random character passwords.

### Task 4

The following is the program output for Fermat and Dixon factorisation of the test data given in “Some test data for factorisation task.docx”

|  |  |  |  |
| --- | --- | --- | --- |
| Fermat | Timing | Dixon | Timing |
| Fermat factorisation of n = 224573= x\*y = 71 \* 3163 | 8ms | Dixon factorisation of n = 224573= x\*y = 3163 \* 71 | 267ms |
| Fermat factorisation of n = 299203= x\*y = 433 \* 691 | 1ms | Dixon factorisation of n = 299203= x\*y = 433 \* 691 | In 22ms |
| Fermat factorisation of n = 1963867= x\*y = 941 \* 2087 | 0ms | Dixon factorisation of n = 1963867= x\*y = 2087 \* 941 | In 1544ms |
| Fermat factorisation of n = 6207251= x\*y = 857 \* 7243 | 10ms | Dixon factorisation of n = 6207251= x\*y = 7243 \* 857 | 1758ms |
| Fermat factorisation of n = 14674291= x\*y = 2267 \* 6473 | 1ms | Dixon factorisation of n = 14674291= x\*y = 6473 \* 2267 | 16150ms |
| Fermat factorisation of n = 23128513= x\*y = 3821 \* 6053 | 0ms | Dixon factorisation of n = 23128513= x\*y = 3821 \* 6053 | 17911ms |
| Fermat factorisation of n = 254855791= x\*y = 509 \* 500699 | 47ms | Dixon factorisation of n = 254855791= x\*y = 500699 \* 509 | 338407ms |
| Fermat factorisation of n = 428279361= x\*y = 16321 \* 26241 | 0ms | Dixon factorisation of n = 428279361= x\*y = 22541019 \* 19 | 701508ms |
| Fermat factorisation of n = 159649552547= x\*y = 346201 \* 461147 | 13ms |  |  |
| Fermat factorisation of n = 189061250479= x\*y = 372299 \* 507821 | 5ms |  |  |
| Fermat factorisation of n = 2211744201787= x\*y = 500699 \* 4417313 | 157ms |  |  |
| Fermat factorisation of n = 7828669742987= x\*y = 2000003 \* 3914329 | 22ms |  |  |
| Fermat factorisation of n = 48560209712519= x\*y = 6850049 \* 7089031 | 0ms |  |  |
| Fermat factorisation of n = 35872004189003= x\*y = 4327423 \* 8289461 | 70ms |  |  |
| Fermat factorisation of n = 737785058178599= x\*y = 26657329 \* 27676631 | 2ms |  |  |
| Fermat factorisation of n = 576460921650883= x\*y = 18983639 \* 30366197 | 98ms |  |  |
| Fermat factorisation of n = 1957432135202107= x\*y = 42345469 \* 46225303 | 6ms |  |  |
| Fermat factorisation of n = 2450609331732137= x\*y = 37164473 \* 65939569 | 330ms |  |  |

The table shows that where the Fermat function was able to factorise all test cases in a maximum of 330ms. Dixons was unable to factorise the test cases 159649552547 or larger. I believe this may be due to a bug in my code that I have not solved it could also be due to my selection of an optimal base. The time taken for Dixons also increases as the number to factorise increases.

Conclusion

The tasks required by the module provided an adequate challenge to push me to learn more about the areas they were based on and the programming practices they implemented. The code for all tasks could have been neatened up to bring them to a professional standard. Task 4 suffers particularly messy code because of poor time management. Thus better time planning should be conducted in future projects.