Assignment 3

Group 3

Joakim Abdinur Iusuf *joakimai@kth.se*
Asta Olofsson *astaol@kth.se*
Anna Skantz *askantz@kth.se*
Lisa Balzar *balzar@kth.se*

February 23, 2022

Contents

Project

Name: TheAlgorithms/Java

URL: https://github.com/TheAlgorithms/Java

Onboarding experience

javaparser

The first project we looked at was javaparser. Since it is a java library the code does not have a main function. There are only tests. The project uses the java build tool maven, which made it easy to run all the test. However, the project was big, over 100k lines of code. Because of this it couldn't be opened in Intellij which some of our team members used. It was also a bit too hard to find suitable functions with high CC for some group members since it was such a big project. Therefore, we had to move on to another project.

TheAlgorithms/Java

On the github page there is not much information about how the project is run. It only says that you can run and edit all the algorithms. After looking at the code it is evident that every algorithm has it's own class, which has a main method. So you run the algorithm you want by compiling and running the class it is in. When it comes to the tests, the project uses the java build tool maven. This could easily be discovered by looking at the files and locating the POM file.

Complexity

The results for the eight complex functions

The CC measurements take exceptions into account.

1. **Encrypt function in HillCipher.java**
   The function implements the polyalphabetic substitution cipher called Hill Cipher. To encrypt a message, each block of n letters is multiplied by an invertible n $\times$ n matrix, against modulus 26. The outcome of the function is clear from the documentation. Both Lizard and manual count for CC was 10 and the LOC of the function was 44, four times the average LOC.

2. **IsSorted function in LinkList_Sort.java**
   The purpose of the function is to sort linked lists given a sorting technique (merge sort, insertion sort or heap sort) and to verify whether the resulting linked list is correctly sorted by comparing the result to an array sorted with the Arrays.sort() native library call. The code for evaluating the insertion sort technique compares however the result with an unsorted array and should therefore return false. It is unclear why this is implemented this way, but it is the intended outcome according to the documentation. The documentation is clear otherwise. Lizard reports 16 CC whereas manual count was 10. The LOC of the function is 92, 8.7 times the average LOC.

3. **addEdge function in AdjacencyListGraph.java**
   The purpose of the function is to check if an edge exists between two vertices that are passed to it. If an edge exists, it returns false, otherwise it returns true. Also, if the one or both of the vertices that are passed to the function don't exist, it also creates the nodes and add

them to the graph. The funtion isn't particularly long, but lizard and a manual count (from more than one group member) show a CC of 8. The documentation is clear.

4. **isBalancedIterative function in CheckIfBinaryTreeBalanced**
   The purpose of the function is to check if a tree is balanced, which in this case is defined as the height of the two subtrees never differing by more than one. Lizard and manual counting gave a CC of 10. The function is very long (78 LOC), but the documentation is fairly clear. I would like the definition of a balanced tree being written in the java documentation, but it's defined inside the function. Almost every line has a comment associated to it.

5. **postorder function in BSTIterative.java**
   The function prints out a binary search tree iteratively in postorder traversal, using one stack. The CC of the function is 8 according to lizard, but when calculated manually, we got 7. The function has 28 LOC. The only documentation that exists is a description that it prints postorder traverself of a binary search tree.

6. **rotated function in HowManyTimesRotated.java**
   The function finds out how many times an array has been rotated from its original sorted state (one rotation being that the first element is put last). The documentation for this function was relatively clear. It describes the problem and the logic behind the solution. It is stated that the position of the minimum element gives you the number of times the array has been rotated. Binary search is used to find the minimum element. The CC of the function is 8 according to lizard, and that's what we got when calculating manually. The function has 16 LOC.

7. **LPS function in LongestPalidromicSubstring.java**
   The function uses dynamic programming to calculate the longest palindromic substring in a string. The CC for the function is 11 according to lizard. When we calculated manually we got a CC of 10. The function has 33 LOC. For this function, there was no documentation. A link was pasted at the top the class claiming to have an explanation of the algorithm, however I cannot find it on the page it links to.

8. **delete in BSTRecursive**
   The function is recursive and deletes a node in a BST. The CC for the function is 9 according to lizard. When we calculated manually we got a CC of 9. The function has 26 LOC. The documentation is clear.

Refactoring

Plan for refactoring complex code

Refactor plan for LPS function in LongestPalindromicSubstring class

The LPS function uses dynamic programming to calculate the longest palindromic substring in a string. The dynamic programming step goes like this: If we have a substring which is a palindrome (look it up in the matrix) and the letter before and after the substring are the same, we have a palindrome.

There are two nestled for-loops in which the dynamic programming matrix is filled. Inside there are some if-statements. Five different branches can be taken. One of them checks for the base case for palindromes of odd length (palindromes of length 1), and another one checks for the base case for palindromes of even length (palindromes of length 2). The functions complexity can be reduced by breaking these two base cases out of the nestled for-loop and fill the matrix with these values before the nestled for-loop. This will be done in a separate function called something in the lines of "initMatrix" so that it will be easy to understand what it does. When this is done we can hopefully refactor the code inside the nestled for-loops so that they contain only one if-else statement. One branch for when the dynamic programming matrix value should be set to true, and another for when it should be false.

The estimated impact of the refactoring is that the CC will be lowered. There will be no difference in complexity since we can use a single for-loop to go through the rows which is going to be initialized. Since we break the function into two functions we have to either make the matrix a class variable, or pass it between functions. The matrix will no longer be scoped to the LPS function, which can be seen as a drawback.

After the refactoring was carried out the LPS function's CC went from 10 to 5. The "initMatrix" function has a CC of 3. As we can see the overall CC dropped with 2.

Refactor plan for the isSorted function in LinkList_Sort class

The isSorted function has a lot of code duplication and is therefore far too complex than it has to be and thus easily refactored. The function consists of three switch cases which all include the same steps essentially:

1. convert array a to linked list
2. sort the linked list according to the specified sorting technique
3. either sort or do not sort the array b that should be compared with the sorted linked list
4. store the sorted linked list into array a
5. compare the two arrays a and b and return true if they are equal, false otherwise

The steps numbered 1, 4, and 5 are almost identical except for some differentiating variable names in the tree switch cases. A plan for refactoring this function would therefore be to move these parts of the code outside the switch cases. Step number 1 of converting the array to a linked list would also be well suited for inserting in a helper function.

The implemented tests for this function revealed an error in the code. This error did probably go unnoticed due to the unnecessary high CC making the code less readable. It is stated in a comment that the insertion sort technique should return false as the array that is used for validating the sorted linked list is never sorted. However, the array gets sorted either way as the array is not a copy of the other array, but a reference to the same array. Because of this, the plan for the refactoring is to also correct the function so that it operates correctly, i.e. returns false for the insertion sort option.

The estimated impact of this refactoring is that the CC will be lowered and that there will be much less LOC in the function. The refactoring is also expected to produce more readable code.

After the refactoring was carried out the CC was reduced by 70% from 10 to 3 and the LOC was reduced by 65% from 92 to 32. The refactoring plan was followed except for additionally implementing a help function for step number 4.

Refactor plan for addEdge function in AdjacencyListGraph class

We would describe the following for-loop in addEdge as essential complexity:

```
for (Vertex v : verticies) {
        if (from.compareTo(v.data) == 0) {
            fromV = v;
        } else if (to.compareTo(v.data) == 0) {
            toV = v;
        }
        if (fromV != null && toV != null) {
            break; // both nodes exist so stop searching
        }
  }
```

Checking if vertices exist in a custom made data structure is complex, and this logic is necessary. The only improvement that can be made is changing

```
if (fromV != null && toV != null) {
    break; // both nodes exist so stop searching
}
```

to

```
if (fromV != null && toV != null) {
     return fromV.addAdjacentVertex(toV);
}
```

as the code under this if-statement does not need to be checked. If we know that both vertices exist, we simply check if there exists an edge between them or not. If an edge exists, we will return false, otherwise we will return true. This decreases the complexity by 1, as we have an extra exit point.

The following if-statements, however, are unnecessary:

```
if (fromV == null) {
    fromV = new Vertex(from);
    verticies.add(fromV);
}

if (toV == null) {
    toV = new Vertex(to);
    verticies.add(toV);
}
```

The code does the exact same thing, so we can minimize duplication by extracting the following helper function:

```
private Vertex createVertex(Vertex vertexToAdd, E vertex) {
    if (vertexToAdd == null) {
        vertexToAdd = new Vertex(vertex);
        verticies.add(vertexToAdd);
    }
    return vertexToAdd;
}
```

This helper can be called like this:

```
fromV = createVertex(fromV, from);
toV = createVertex(toV, to);
```

This decreases the complexity by 2, as we have removed two if-statements. The CC of addEdge was previously 8 and refactoring the method in this way would decrease the cyclomatic complexity to 5. This is a 37,5% decrease in CC.

Refactor plan for postorder function in BSTIterative class

The function prints out a binary search tree iteratively in postorder traversal, using one stack. The function does this by moving down to the leftmost node, and in the meantime push the nodes to a stack. When leaf is reached, the stack is checked for right children. The function contains several while-loops and if and else statements in order to traverse to leaves. The CC is currently 7. The algorithm is relatively complicated and the high complexity is somewhat justified. The code is relatively long and difficult to follow. It is however possible to extract parts of the function into a separate one. Half of the function is in an else block. This else block is reached when the leftmost node is reached, and elements from the stacks are popped. This whole block can be moved to a separate function. From this, you can also extract another block of that function to a separate one. Doing this would reduce the CC to 4, resulting in a 43% decrease.

Carried out refactoring

Refactored methods

Refactorings added by Asta:

- LPS in LongestPalindromicSubstring.java. CC was reduced by 50% from 10 to 5.
- subSet in MinimumSumPartition.java CC was reduced by 44% from 9 to 4.

You can find the refactored code on this branch: `https://github.com/AnnaSkantz/Java/tree/AstaRefactor`

Refactorings added by Anna:

- encrypt in HillCipher.java, CC was reduced by 60%, from 10 to 4.
- decrypt in HillCipher.java, CC was reduced by 60%, from 10 to 4.

- isSorted in LinkList_Sort, CC was reduced by 70% from 10 to 3.

The refactored code can be found at https://github.com/AnnaSkantz/Java/tree/AnnaRefactor

Refactorings added by Joakim:

- addEdge in AdjacencyListGraph. CC was reduced by 37,5%, from 8 to 5.
- isBalancedIterative in CheckIfBinaryTreeBalanced. CC was reduced by 40%, from 10 to 6.

The refactored addEdge function can be found at https://github.com/AnnaSkantz/Java/tree/issue%235

The refactored isBalancedIterative function can be found at https://github.com/AnnaSkantz/Java/tree/issue%236

Refactorings added by Lisa:

- postorder in BSTIterative. CC was reduced by 43%, from 7 to 4.
- rotated in HowManyTimesRotated. CC was reduced by 37,5%, from 8 to 5.

The refactored code can be found on this branch: https://github.com/AnnaSkantz/Java/tree/LisaRefactor

Coverage

Tools

We used Jacoco [1] and OpenClover [2].

OpenClover

OpenClover's documentation was easy to follow and you could add it globally in your maven installation, which means you don't have to change anything in the local POM file for it to work. However, it did not work for all of our team members. After following the quick start guide only one of our team member got it to work as expected. It produces a report for everyone, but it was empty for 3 out of 4 team members because it couldn't record the tests.

Jacoco

Jacoco is easy to work with. You simply add a plugin in the POM file and it produces a report on running mvn test. Compared to OpenClover all of our team members got it to work.

Our own coverage tool

For this assignment we decided that everyone should try to do their own coverage tool for their functions. This resulted in two different kinds of coverage tools. One kind is run from the main function, and one that is run from the function. Here is the link to the main method version `https://github.com/AnnaSkantz/Java/tree/LisaTests`. The patch can be seen

[1] `https://www.jacoco.org/jacoco/`
[2] `https://openclover.org/`

using the following command:

```
git diff oldMaster src/main/java/com/thealgorithms/datastructures/trees/BSTIterative.java
```

To run it, just compile and run the BSTIterative class. The results will be printed out. Here is the link to the test version https://github.com/AnnaSkantz/Java/tree/AnnaTests. The patch can be seen using the following command:

```
git diff oldMaster src/main/java/com/thealgorithms/ciphers/HillCipher.java
```

To run it, run the tests. The results can be found in the file /tmp/coverageInfo.txt.

The one that is run from the main function works like this. You specify what code you want to run in main, and the tools prints to standard out which branches was reached in that code. One downside to this method is that it cannot check what branches are taken in the tests. It can only check the code which is run from main.

The other method that is run from the function you want to test works like this. It records the branches taken and writes their ID's to a file when the function exits. Since everything happens in the function it can record the branches taken in the tests, as well what branches are being taken when some code is run in main. One downside however is that you have to write to a file in the function before the exit points.

Both of the tools have accurate measurements for the functions that they are implemented on, since we write to the datastructure for every branch. As long as the one who added the code in the function has not missed a branch. One great downside to both of the tools is that you have to add code in the function you want to test code coverage for. This makes the real code harder to read. It could also make it slower, since you have to print things out and write to files. If you compare that with for example Jacoco or OpenClover which you don't have to add a bunch of code for it to work it is very inefficient and hard to use. Especially if you want to look at code coverage for bigger projects.

Coverage improvement

The branch coverage of the project was increased from 1% to 5%.

Report of old coverage: https://github.com/AnnaSkantz/Java/tree/master/jacocoCoverageReportBefore

Report of new coverage: https://github.com/AnnaSkantz/Java/tree/master/jacocoCoverageReportAfter

Test cases added

Tests added by Asta:

1. LongestPalindromicSubstringTest.java
   - oddLetter7PalindromeTest
   - evenLetterPalindromeTest
   - noPalindromeTest
   - nullStringTest
   - emptyStringTest
2. BSTRecursiveTest.java
   - removeFromEmptyTree
   - removeNodeWithTwoChildrenTest
   - addAndRemoveOneElementTest
3. MinimumSumPartition.java
   - minimimSumPartitionTest

Tests added by Anna:

1. LinkList_SortTest.java

- testThatisSortedReturnsTrueWhenMergeSort
- testThatisSortedReturnsFalseWhenInsertionSort
- testThatisSortedReturnsTrueWhenHeapSort

2. HillCipherTest.java
    - testThatEncryptionIsCorrect

Tests added by Joakim:

1. AdjacencyListGraphTest.java
    - testEdgeExistsInAddEdge
    - testEdgeDoesNotExistWhenToIsNull
    - testEdgeDoesNotExistWhenFromIsNull
    - testEdgeDoesNotExistWhenFromAndToAreNull

Tests added by Lisa:

1. BSTIterativeTest.java
    - testPostorderPrintMessageTree
    - testPostorderPrintMessageEmptyTree
    - testPreorderPrintMessageTree
    - testPreorderPrintMessageEmptyTree
    - testAddAndRemoveNode

Number of test cases added: two per team member (P) or at least four (P+).

Self-assessment: Way of working

The state that best describes us is *Working well*. We are making progress as planned, and are almost done with the task. We naturally apply practices, many of which were established in previous assignments. The tools we are using, e.g. GitHub support the way we work, and we are continually making use of our practices and tools. The self-assessment is almost unanimous, as three of us think we are in a state of *working well* and one member thinks *in place* best describes us. The reason, which we all agree on, is that we are naturally applying the practices, but we need to think about how we do it. The reason for this is most likely that we are using Git in a slightly different way, so the work flow isn't as smooth as in assignment 1 and 2.

We have improved since assignment 1 though. Everyone is more comfortable using Git, how to write commit messages, and collaborate using version control. One thing we could have done better this assignment is decide beforehand how we would use GitHub, e.g. how to name branches.

Overall experience

One thing we learned is that getting code coverage tools to work on large projects isn't easy. One has to configure build files, e.g. pom.xml when using jacoco. When trying to make Open Clover work on the javaparser project, only one of us was successful, despite everyone taking the exact same steps to get the coverage tool to work.

For some of us, getting lizard to work was also difficult. Again, despite everyone taking the exact same steps to make lizard work, some of us still experienced difficulty and had to make additional changes to files to make the tool work. One common denominator is that the person that uses linux found it easier to make Open Clover and lizard to work, where as those of us who work on windows and mac had more difficulties.

Another thing we learned is that working on open source projects does not necessarily force you to understand a larger portion of the system. The functions we worked on only relied on other parts of the class they belonged to. This made it relatively easy to understand how the functions work, and make changes to them. In assignment 4, we will most likely pick a more complex project in which classes and functions are more dependent on each other, so that we gain some experience in working in systems that probably look more like the code bases we will work in once we graduate.

One thing that is worth mentioning is that the quality of the code between different open source projects can differ widely. When we looked at some open source projects, it was really difficult to find large and complex functions, as the code was clean and followed SOLID principles (e.g. each function having a single responsibility). The code in the project we worked in, however, was pretty bad, evidenced by our refactorings. One team member even found a function that failed tests, and had to change it to make it work properly. Another thing that is worth mentioning is that code coverage can vary widely between different projects, with some projects being tested thoroughly, and others almost having no tests. It also seems that projects with more tests tend to have cleaner code with fewer complex functions, whereas projects with few tests also have more complex and less readable code.