

# P1 – Search in Late-Binding Solitaire

## Overview

For this practical, the requirement was to write a program that can play Accordion. It should be able to check whether a given solution is correct, check whether a game is solvable or not given an initial layout and also do both the above for a variant of the original game called Accordion with saving grace. I was managed to attempt the first part, the second part and the first half of the third part. However, my program fails to run successfully for every input that it was given. There are several input cases it is able to solve and generate the correct answer but unfortunately several where it is unable to. Therefore, in this report I aim prove that my program does work as it should in given situations, but also demonstrate where it doesn't and try to discuss why that might be the case.

## Design

When it comes to the design of my solution, I took a sort of top-down approach when designing the structure of the program as well as the data associated with it. The design consists of layers, where each layer builds onto of the previous one, thus it is hierarchal in nature. The lowest layer is the cards involved in the game's initial layout. Each stores information regarding the suit and card number. This is the basic building block for the entire program and is static. Then comes the states which builds on top of the cards layer. Each state contains the cards involved during a particular state of the game, that is, after a move has been made. Lastly, a stack is used as the top layer and store all the states of the game. Figure 1 gives a visual representation of what this abstraction would look like.

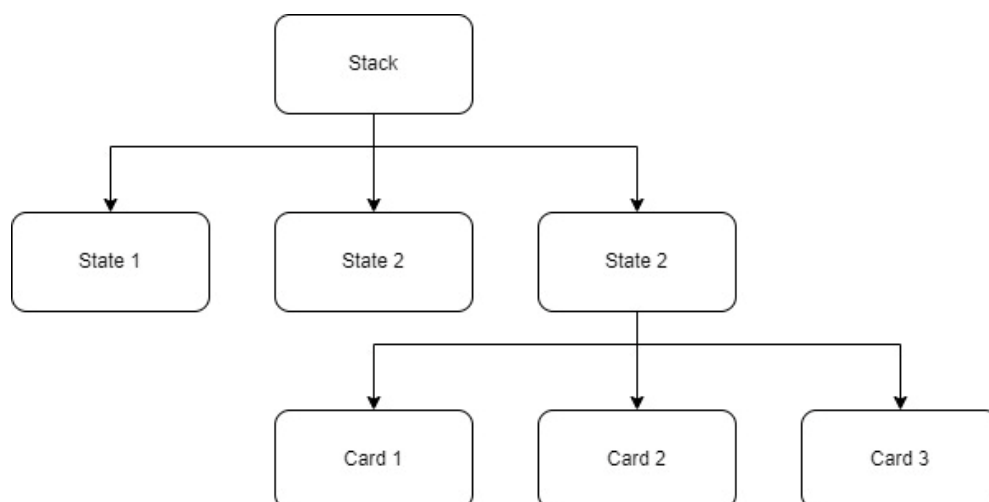


Figure 1: a top-down diagram showing the structure of the program

## Implementation

The cards layer is implemented using a Card class. This is a simple method which has a field for the suit, card number and an additional number which is used later when comparing two cards after a move. The class also has a constructor for initialization and each field has its own getter methods.

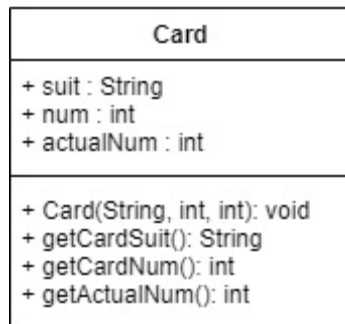


Figure 2: a class diagram of the Card class

The States layer is implemented through the State class, where each state stores the current layout of cards, moves that have been made to reach the current moves and all the possible moves that can be made from the current state. Additionally, this class has function that play a key role in finding a solution to a given game. The possibleMoves() function goes through the list of cards in the current state and finds all the possible moves that can be made. The values are then stored in the possible moves list. This is done by starting from the last card in the layout and looping until the first card is reached. Each time a card is accessed, the card one move to the left is also accessed. This is the deck to potentially move to, so its suit and number are compared to the suit and number of the card that's being moved. If they match, then the deck number of both cards are added to the possible moves list. The same steps are repeated for the card that's three moves to the left. This function is accompanied by the updatePossibleMoves() function which updates the list of possible moves after a move has been made by removing the start and end deck numbers of the move. There are three more relevant functions which deal with certain checks. The hasNextMove() function checks whether the state has anymore moves left by checking whether the possible moves list is empty or not. The solutionFound() function checks whether the game has been completed by checking whether the list of cards in the current state has a size of one. Lastly, getNumberOfMoves() function is returns the number of moves that have been made by dividing the size of the moves made list by two. The other functions in this class are basically helper functions.



Figure 3: a class diagram for the States class

The stack is implemented in the StateStack class. The stack is implemented as a list and has functions for providing all the basic stack operations such push, pop and peek, just like every other generic stack class.

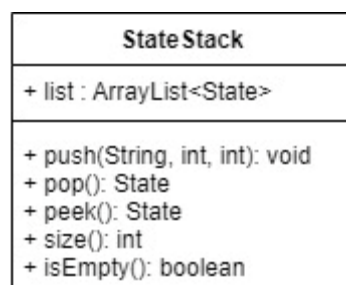


Figure 4: a class diagram for the StateStack class

The final part of the implementation, and the most important, is the LBSMain class which contains the actual algorithms for coming up with a solution for each part. This class has a function to read the command-line input, a function to print an error message and a main method which is where the algorithms are, and so will be the focus.

For part one, which is dealt within the "SOLVE" case, the requirement is to check whether a given solution to a game is correctly solves it. To do that, the initial layout of integers is first converted into a list of Card objects. This is done through a loop which goes through each card and calculates the suit and number of each card, and then these values are sent to the Card class to create a card object. To determine the suit and number of a card, a conditional is used where each branch has a range of numbers (as defined in the specification) and if the actual number read from the layout is within that range, it enters that branch. Each

range identifies a particular suit, so a value for its suit is automatically assigned. For example, if the number read from the layout is between 1 and 13, then it goes to the first branch where it is assigned as spades. The card number is then calculated by the specified number in the branch from the actual card number. Doing this would allow all of the cards, regardless of their suit, to have a number between 1 and 13, which makes it easier when comparing two cards. Each of the card object is then stored as a list which is identical to the original layout list.

The next step is to make the moves, as specified from the input read, on the card object list. In a loop, a pair of values is read from the moves list, where the first value is the card to move, and the second value is that deck number to move it to. In another loop, the card to move is compared with each card in the card object list, until the right card is found. When this happens the deck number of that card is retrieved. The last step is to check whether the given move to be made is valid. This is done using a conditional statement which checks whether the distance between the two is valid and another conditional statement which checks whether either the suit or the number matches. These values are accessed from the card object which has get methods for each of its fields. If so the move is made and these steps are repeated for the next move until the end of the move list is reached. Otherwise, the move made is invalid and the loop is terminated. At the end, a final conditional statement checks whether the number of decks at the end equates to one and outputs "true" if it does since this is a correct solution. If not, then the solution is incorrect and "false" is displayed. For this part of the specification the state and stack classes aren't used since they are irrelevant here.

The second part is contained within the "SOLVE" case. Here the requirement is to take a given layout and find a solution if it exists. The first step of this is the same as for part one. After generating the card object layout, this list and a list of the moves made (which will be empty) is sent to the state class to create the initial state of the game. This state is then pushed into the stack before entering the main loop which terminate either when a move has been found or when there aren't any more moves left. The initial state is popped out of the stack and a conditional statement is used after this to check whether the this state has solved the game by using the solutionFound() function. If it has then the current state has the solved layout, then the loop is exited. If not, then another conditional statement checks if the state has anymore moves left using the hasNextMoves() function. If it doesn't, then this state is discarded and it goes to the end of the loop where a check is done to see whether all of the remaining states in that stack have at least one move to be made. If not there aren't any moves left so the loop terminates. If it still does, then it loops back to the start of the loop and the next state at the top of the stack is popped. The previous steps are done when hasNextMove() returns false. But if it returns true, then the three of its lists are accessed (card object layout, moves made and possible moves). The next move to be made is accessed from the possible moves list, which will again be a pair of values. But this time, instead of the first value being the actual card number, it contains the deck number to make the process of making the move easier. These values are immediately removed afterwards since we don't want to make the same move repeatedly the next time a move is made from this state since it would result in an infinite loop. This has to be updated in the current state

so the `updatePossibleMoves()` function is called and the updated list is passed. Following this, the pair of values are added to the empty moves made list and the actual move is made on the card object layout list. This is done using a `set()` function where the deck number to move to and the card object to move is passed. This does a copy of the specified card to the specified deck. So the `remove()` function is then called to remove the card object from the previous deck number that it was in. A new state is then created where the new card object list and moves made list is passed. The previous state is then pushed back onto the stack followed by the new state. The above steps then repeat. Once the main loop terminates, the appropriate output is displayed based on whether it terminated because a solution has been found or because there aren't any moves left. The number of moves displayed by calling the `getNumberOfMoves()` function.

The last part of my implementation is the check grace part of part three of the requirements. I didn't get to implement the solve grace part, so it won't be discussed here. This check follows the same steps as the check from part one. The only difference is in the part where the check for whether a valid move has been made. The same conditional statements from the other check is nested inside another conditional statement which checks that there hasn't been a second grace move. To check for this, a third conditional statement is included along with the other two which checks whether the move that has just been made was a grace move. If so, a Boolean is set to true.

## Testing

I tested my program by using stacscheck as well as running a few of my own tests. Unfortunately, due to my program not functioning properly, it wasn't able to pass all the tests from stacscheck. However, when I tested it on own, there were better results as my program was able to pass more of those tests. My approach was to have a folder with a bunch of test files for each part. Then during the testing process I transferred each file to the source directory of my program to test each input case individually. Most of these files are directly from stacscheck.

### Tests for Part 1:

- Input: a list of moves for the given layout (test1.tx and test2.txt)
- Expected output: true
- Actual output: none (error)

```
-  
[9, 0, 8, 0, 23, 1, 36, 1]  
[9, 0, 8, 0, 23, 1, 36, 1, 41, 2]  
-  
Unsolvable:  
-1  
sb409@pc5-002-l:~/Documents/Third_Year/CS3015/P1-LateBinding/src $ javac LBSMain.java  
sb409@pc5-002-l:~/Documents/Third_Year/CS3015/P1-LateBinding/src $ java LBSMain CHECK test1.txt test2.txt  
[3, 17, 1, 17, 0, 30, 0]  
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 0 out of bounds for length 0  
    at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)  
    at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)  
    at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:266)  
    at java.base/java.util.Objects.checkIndex(Objects.java:359)  
    at java.base/java.util.ArrayList.get(ArrayList.java:427)  
    at LBSMain.main(LBSMain.java:306)  
sb409@pc5-002-l:~/Documents/Third_Year/CS3015/P1-LateBinding/src $
```

### Tests for Part 2:

- Input: a layout with one card (test1.txt)
- Expected output: 0
- Actual output: 0

```
PROBLEMS 10 OUTPUT DEBUG CONSOLE TERMINAL  
Unsolvable:  
-1  
sb409@pc5-002-l:~/Documents/Third_Year/CS3015/P1-LateBinding/src $ javac LBSMain.java  
sb409@pc5-002-l:~/Documents/Third_Year/CS3015/P1-LateBinding/src $ java LBSMain SOLVE test.txt  
clubs  
12  
Solution found:  
0  
sb409@pc5-002-l:~/Documents/Third_Year/CS3015/P1-LateBinding/src $ javac LBSMain.java  
sb409@pc5-002-l:~/Documents/Third_Year/CS3015/P1-LateBinding/src $ java LBSMain SOLVE test.txt  
Solution found:  
0  
sb409@pc5-002-l:~/Documents/Third_Year/CS3015/P1-LateBinding/src $
```

- Input: a solvable layout with four cards (test2.txt)
- Expected output: 3
- Actual output: 3

```
PROBLEMS 10 OUTPUT DEBUG CONSOLE TERMINAL  
Solution found:  
0  
sb409@pc5-002-l:~/Documents/Third_Year/CS3015/P1-LateBinding/src $ javac LBSMain.java  
sb409@pc5-002-l:~/Documents/Third_Year/CS3015/P1-LateBinding/src $ java LBSMain SOLVE test.txt  
[30, 2]  
-  
[30, 2, 16, 0]  
-  
[17, 1]  
-  
[17, 1, 30, 1]  
-  
[17, 1, 17, 0]  
-  
[17, 1, 17, 0, 30, 0]  
-  
Solution found:  
3  
sb409@pc5-002-l:~/Documents/Third_Year/CS3015/P1-LateBinding/src $
```

- Input: a solvable layout with five cards (test3.txt)
- Expected output: 4
- Actual output: 4

```
PROBLEMS 10 OUTPUT DEBUG CONSOLE TERMINAL

[17, 1, 17, 0, 30, 0]
-
Solution found:
3
sb409@pc5-002-l:~/Documents/Third Year/CS3015/P1-LateBinding/src $ javac LBSMain.java
sb409@pc5-002-l:~/Documents/Third Year/CS3015/P1-LateBinding/src $ java LBSMain SOLVE test.txt
[1, 1]
-
[1, 1, 41, 2]
-
[1, 1, 41, 0]
-
[1, 1, 41, 0, 2, 1]
-
[1, 1, 41, 0, 2, 1, 2, 0]
-
Solution found:
4
sb409@pc5-002-l:~/Documents/Third Year/CS3015/P1-LateBinding/src $
```

- Input: a solvable layout with 17 cards (test4.txt)
- Expected output: 16
- Actual output: 16

```
PROBLEMS 10 OUTPUT DEBUG CONSOLE TERMINAL


[32, 15, 48, 11, 8, 10, 13, 6, 13, 3, 8, 5, 19, 6, 32, 6, 13, 0, 30, 2, 8, 0, 32, 2, 6, 0, 35, 1, 48, 1]
-
[32, 15, 48, 11, 8, 10, 13, 6, 13, 3, 8, 5, 19, 6, 32, 6, 13, 0, 30, 2, 8, 0, 32, 2, 6, 0, 32, 0]
-
[32, 15, 48, 11, 8, 10, 13, 6, 13, 3, 8, 5, 19, 6, 32, 6, 13, 0, 30, 2, 8, 0, 32, 2, 6, 0, 32, 0, 48, 1]
-
[32, 15, 48, 11, 8, 10, 13, 6, 13, 3, 8, 5, 19, 6, 32, 6, 13, 0, 30, 2, 8, 0, 32, 2, 6, 0, 32, 0, 35, 0]
-
[32, 15, 48, 11, 8, 10, 13, 6, 13, 3, 8, 5, 19, 6, 32, 6, 13, 0, 30, 2, 8, 0, 32, 2, 6, 0, 32, 0, 35, 0, 48, 0]
-
Solution found:
16
sb409@pc5-002-l:~/Documents/Third Year/CS3015/P1-LateBinding/src $
```

- Input: a solvable layout with 24 cards (test5.txt)
- Expected output: 23
- Actual output: 23

```
PROBLEMS 10 OUTPUT DEBUG CONSOLE TERMINAL

[21, 20, 6, 21, 6, 18, 4, 18, 21, 16, 4, 17, 21, 15, 21, 14, 4, 12, 21, 13, 21, 10, 16, 10, 16, 9, 4, 7, 16, 8, 4, 6, 4, 5]
-
[21, 20, 6, 21, 6, 18, 4, 18, 21, 16, 4, 17, 21, 15, 21, 14, 4, 12, 21, 13, 21, 10, 16, 10, 16, 9, 4, 7, 16, 8, 4, 6, 4, 5, 4, 4]
-
[21, 20, 6, 21, 6, 18, 4, 18, 21, 16, 4, 17, 21, 15, 21, 14, 4, 12, 21, 13, 21, 10, 16, 10, 16, 9, 4, 7, 16, 8, 4, 6, 4, 5, 4, 4, 16, 2]
-
[21, 20, 6, 21, 6, 18, 4, 18, 21, 16, 4, 17, 21, 15, 21, 14, 4, 12, 21, 13, 21, 10, 16, 10, 16, 9, 4, 7, 16, 8, 4, 6, 4, 5, 4, 4, 16, 2, 4, 3]
-
[21, 20, 6, 21, 6, 18, 4, 18, 21, 16, 4, 17, 21, 15, 21, 14, 4, 12, 21, 13, 21, 10, 16, 10, 16, 9, 4, 7, 16, 8, 4, 6, 4, 5, 4, 4, 16, 2, 4, 3, 4, 0]
-
[21, 20, 6, 21, 6, 18, 4, 18, 21, 16, 4, 17, 21, 15, 21, 14, 4, 12, 21, 13, 21, 10, 16, 10, 16, 9, 4, 7, 16, 8, 4, 6, 4, 5, 4, 4, 16, 2, 4, 3, 4, 0, 16, 1]
-
[21, 20, 6, 21, 6, 18, 4, 18, 21, 16, 4, 17, 21, 15, 21, 14, 4, 12, 21, 13, 21, 10, 16, 10, 16, 9, 4, 7, 16, 8, 4, 6, 4, 5, 4, 4, 16, 2, 4, 3, 4, 0, 17, 0]
-
[21, 20, 6, 21, 6, 18, 4, 18, 21, 16, 4, 17, 21, 15, 21, 14, 4, 12, 21, 13, 21, 10, 16, 10, 16, 9, 4, 7, 16, 8, 4, 6, 4, 5, 4, 4, 16, 2, 4, 3, 4, 0, 17, 0, 16, 0]
-
Solution found:
23
sb409@pc5-002-l:~/Documents/Third Year/CS3015/P1-LateBinding/src $
```

- Input: a solvable layout containing 8 cards (test8.txt)
- Expected output: 7
- Actual output: -1

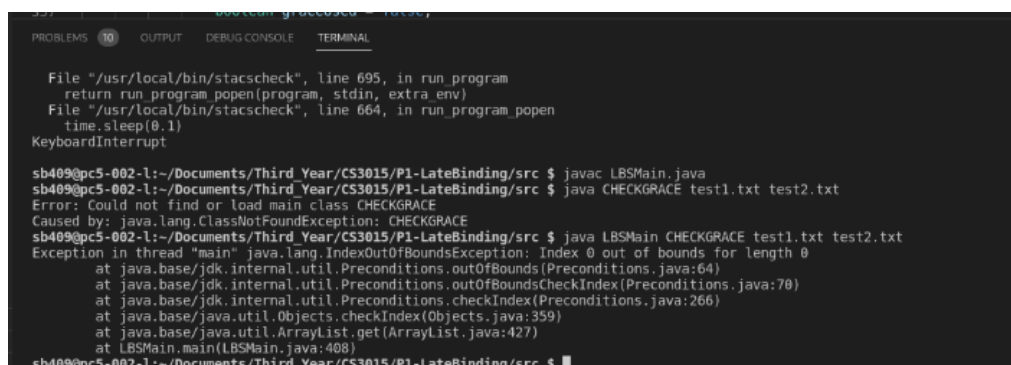


```
PROBLEMS 10 OUTPUT DEBUG CONSOLE TERMINAL
[9, 0, 8, 0, 41, 4, 23, 1, 36, 1]
-
[9, 0, 8, 0, 36, 2]
-
[9, 0, 8, 0, 36, 2, 41, 3]
-
[9, 0, 8, 0, 23, 1]
-
[9, 0, 8, 0, 23, 1, 41, 3]
-
[9, 0, 8, 0, 23, 1, 41, 3, 36, 1]
-
[9, 0, 8, 0, 23, 1, 36, 1]
-
[9, 0, 8, 0, 23, 1, 36, 1, 41, 2]
-
Unsolvable:
-1
sb409@pc5-002-l1:~/Documents/Third Year/CS3015/P1-LateBinding/src $
```

- Input: an unsolvable layout (test7.txt)
- Expected output: -1
- Actual output: none (infinite loop)

### Tests for Part 3:

- Input: a list of moves for a given layout with saving grace
- Expected output: true
- Actual output: None (error)



```
PROBLEMS 10 OUTPUT DEBUG CONSOLE TERMINAL
File "/usr/local/bin/stacccheck", line 695, in run_program
return run_program popen(program, stdin, extra env)
File "/usr/local/bin/stacccheck", line 664, in run_program popen
time.sleep(0.1)
KeyboardInterrupt

sb409@pc5-002-l1:~/Documents/Third Year/CS3015/P1-LateBinding/src $ javac LBSMain.java
sb409@pc5-002-l1:~/Documents/Third Year/CS3015/P1-LateBinding/src $ java CHECKGRACE test1.txt test2.txt
Error: Could not find or load main class CHECKGRACE
Caused by: java.lang.ClassNotFoundException: CHECKGRACE
sb409@pc5-002-l1:~/Documents/Third Year/CS3015/P1-LateBinding/src $ java LBSMain CHECKGRACE test1.txt test2.txt
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 0 out of bounds for length 0
    at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)
    at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
    at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:266)
    at java.base/java.util.Objects.checkIndex(Objects.java:359)
    at java.base/java.util.ArrayList.get(ArrayList.java:427)
    at LBSMain.main(LBSMain.java:408)
sb409@pc5-002-l1:~/Documents/Third Year/CS3015/P1-LateBinding/src $
```

## Evaluation and Conclusion

From all the testing, it is evident that my program behaves very inconsistently. For some input cases it is clear that it is able to calculate the correct output, but for a lot of the input cases that doesn't seem to be the case. There is a reoccurring pattern that can be observed, which is that the smaller the size of the program and the lesser the possible paths, the more able it is to produce a correct output. So, for really simple problems, my program will most likely produce a reliable result. Whereas larger problems tend to have even more possible paths to choose from which reduces the winnability and increases the hardness. But, this shows a fundamental flaw in the way I decided to implement my solution as it doesn't extend to broader input cases. Another point to note is that, my program is unable to show



that a game is unsolvable. In such cases it ends up going in an infinite loop. All of this relates to the solving section, but when it comes to the checking sections, the program immediately runs into an error as it isn't able to read the layout list for whatever reason. I have gone through several debugging cycles where I managed to fix several issues. However, I wasn't able to completely debug it.

Apart from all the errors, one big thing I could improve on is having an actual heuristic in place. This would drastically improve the performance since my program is currently really inefficient because of how it does an exhaustive search, where it goes through every single possible combination of moves. One other thing would be extending the range of valid card numbers that my program accepts since currently it only accepts a fixed range and isn't very dynamic.

Since my program has several bugs and isn't able to run consistently, I'm unable to properly evaluate its performance. But I decided to evaluate just the situations where my program happens to run and produce an input which is only for the second part.

Layout	Number of states visited	Time taken to solve (ms)
test2.txt	6	1
test3.txt	5	1
test4.txt	121702	30899
test5.txt	24	3