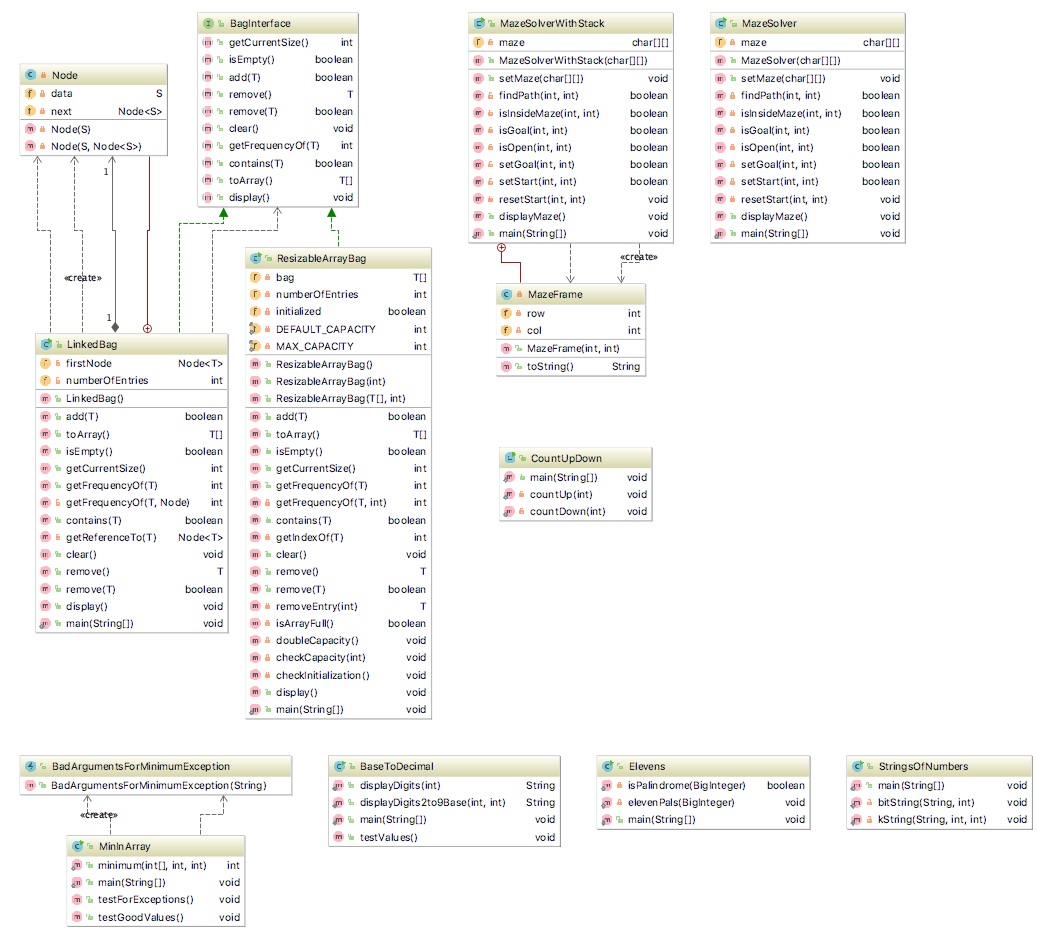
# Comp151 Lab05

In Lab05 you will be working on **six** separate applications:

1. BaseToDecimal
2. StringsOfNumbers
3. Elevens
4. MazeSolver
5. MazeSolverWithStack
6. MinInArray

Each application has a main inside the corresponding .java file and there are no dependencies between them. See the descriptions below. The skeletons for each program are provided.

### UML Diagram:

****

1. **BaseToDecimal**

If n is a positive integer in Java, n % 10 is its rightmost digit and n / 10 is the integer obtained by dropping the rightmost digit from n. Using these facts, write a recursive method called displayDigits that displays the digits of an integer n in decimal. For example, the integer number 345 should be displayed as a String "3 4 5"; where the negative integer number -345 should be displayed as a String "-3 4 5". Note the spaces between the digits.

Now observe that you can display n in any base between 2 and 9 by replacing 10 with the new base. Write displayDigits2to9Base method that is a revised version of yours displayDigits method, to accommodate a given base. Please note that the modified method will essentially convert a decimal number into its equivalent in the given base. For example, 10 in base 8 will be displayed as a String "1 2"; 5 in base 2 will be displayed as a String "1 0 1", and so on.

Test your methods with the driver provided in main.

1. **Elevens**

Write a recursive method that checks the result of sequence of 1s multiplied by the same sequence of 1s for palindrome. In this application, you need to utilize **methods** and **constants** from BigInteger class. Implement the algorithm that you designed as part of your pre-lab.

Your program should produce the following output:

**1 \* 1 is 1 - and it is a PALINDROME  
11 \* 11 is 121 - and it is a PALINDROME  
111 \* 111 is 12321 - and it is a PALINDROME  
1111 \* 1111 is 1234321 - and it is a PALINDROME  
11111 \* 11111 is 123454321 - and it is a PALINDROME  
111111 \* 111111 is 12345654321 - and it is a PALINDROME  
1111111 \* 1111111 is 1234567654321 - and it is a PALINDROME  
11111111 \* 11111111 is 123456787654321 - and it is a PALINDROME  
111111111 \* 111111111 is 12345678987654321 - and it is a PALINDROME  
1111111111 \* 1111111111 is 1234567900987654321 - and it is NOT a PALINDROME**

1. **StringsOfNumbers**

Write two recursive methods that generate string of numbers. The first method generates all the possible strings that contain the combinations of *n* bits, where *n* is given by the user. The second method is generalization of the first method. It also generates all the permutations of *n* numbers, but the numbers are drown from [0..k); where the *k* is given by the user. See sample runs below. Implement the algorithms that you designed as part of your pre-lab.

**SAMPLE RUN:**

**Please enter an integer value of n representing the number of digits in a string**

**3**

**Generating binary-Strings:**

**000**

**001**

**010**

**011**

**100**

**101**

**110**

**111**

**Please enter an integer value k; strings of length n will be drown from 0..k-1**

**4**

**Generating k-Strings:**

**000**

**001**

**002**

**003**

**010**

**011**

**012**

**013**

**020**

**021**

**022**

**023**

**030**

**031**

**032**

**033**

**100**

**101**

**102**

**103**

**110**

**111**

**112**

**113**

**120**

**121**

**122**

**123**

**130**

**131**

**132**

**133**

**200**

**201**

**202**

**203**

**210**

**211**

**212**

**213**

**220**

**221**

**222**

**223**

**230**

**231**

**232**

**233**

**300**

**301**

**302**

**303**

**310**

**311**

**312**

**313**

**320**

**321**

**322**

**323**

**330**

**331**

**332**

**333**

**Process finished with exit code 0**

**SAMPLE RUN WITH DEBUGGING STATEMENTS:**

**Please enter an integer value of n representing the number of digits in a string**

**3**

**Generating binary-Strings:**

**--> FOR DEBUGGING: At the start of the method: str is "" and n is 3**

**----> FOR DEBUGGING: About to call itself to append '0': str is "" and n is 3**

**--> FOR DEBUGGING: At the start of the method: str is "0" and n is 2**

**----> FOR DEBUGGING: About to call itself to append '0': str is "0" and n is 2**

**--> FOR DEBUGGING: At the start of the method: str is "00" and n is 1**

**----> FOR DEBUGGING: About to call itself to append '0': str is "00" and n is 1**

**--> FOR DEBUGGING: At the start of the method: str is "000" and n is 0**

**---> FOR DEBUGGING: Executing base case**

**000**

**----> FOR DEBUGGING: About to call itself to append '1': str is "00" and n is 1**

**--> FOR DEBUGGING: At the start of the method: str is "001" and n is 0**

**---> FOR DEBUGGING: Executing base case**

**001**

**----> FOR DEBUGGING: About to call itself to append '1': str is "0" and n is 2**

**--> FOR DEBUGGING: At the start of the method: str is "01" and n is 1**

**----> FOR DEBUGGING: About to call itself to append '0': str is "01" and n is 1**

**--> FOR DEBUGGING: At the start of the method: str is "010" and n is 0**

**---> FOR DEBUGGING: Executing base case**

**010**

**----> FOR DEBUGGING: About to call itself to append '1': str is "01" and n is 1**

**--> FOR DEBUGGING: At the start of the method: str is "011" and n is 0**

**---> FOR DEBUGGING: Executing base case**

**011**

**----> FOR DEBUGGING: About to call itself to append '1': str is "" and n is 3**

**--> FOR DEBUGGING: At the start of the method: str is "1" and n is 2**

**----> FOR DEBUGGING: About to call itself to append '0': str is "1" and n is 2**

**--> FOR DEBUGGING: At the start of the method: str is "10" and n is 1**

**----> FOR DEBUGGING: About to call itself to append '0': str is "10" and n is 1**

**--> FOR DEBUGGING: At the start of the method: str is "100" and n is 0**

**---> FOR DEBUGGING: Executing base case**

**100**

**----> FOR DEBUGGING: About to call itself to append '1': str is "10" and n is 1**

**--> FOR DEBUGGING: At the start of the method: str is "101" and n is 0**

**---> FOR DEBUGGING: Executing base case**

**101**

**----> FOR DEBUGGING: About to call itself to append '1': str is "1" and n is 2**

**--> FOR DEBUGGING: At the start of the method: str is "11" and n is 1**

**----> FOR DEBUGGING: About to call itself to append '0': str is "11" and n is 1**

**--> FOR DEBUGGING: At the start of the method: str is "110" and n is 0**

**---> FOR DEBUGGING: Executing base case**

**110**

**----> FOR DEBUGGING: About to call itself to append '1': str is "11" and n is 1**

**--> FOR DEBUGGING: At the start of the method: str is "111" and n is 0**

**---> FOR DEBUGGING: Executing base case**

**111**

**Please enter an integer value k; strings of length n will be drown from 0..k-1**

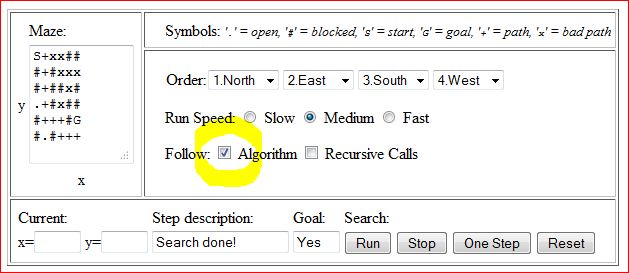
**4**

**Generating k-Strings:**

**[ … ]**

## Maze Solver

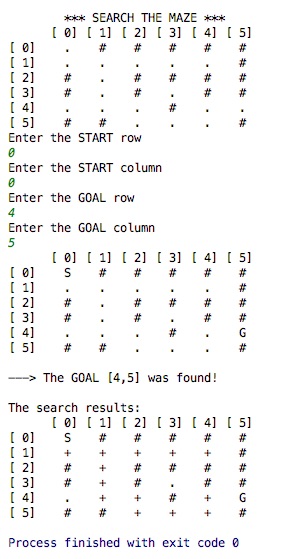
* 1. Go to <https://www.cs.bu.edu/teaching/alg/maze/> and analyze the given algorithm
  2. Run the applet at the bottom of the page with the “Algorithm” box checked:

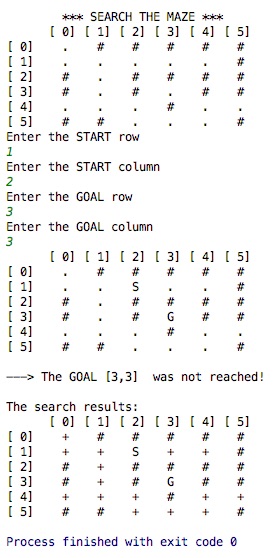


* 1. Implement the following maze search algorithm using recursion, use MazeSolver.java as the starting point (please notice that our algorithm does not unmark [x,y] if it is not in the solution path):

**FIND-PATH(x, y)**

* if ([x,y] outside maze) return false
* if ([x,y] is goal) return true
* if ([x,y] not open) return false
* mark [x,y] as part of solution path
* if (FIND-PATH(North of x,y) == true) return true
* if (FIND-PATH(East of x,y) == true) return true
* if (FIND-PATH(South of x,y) == true) return true
* if (FIND-PATH(West of x,y) == true) return true
* return false
  1. See two sample runs below:

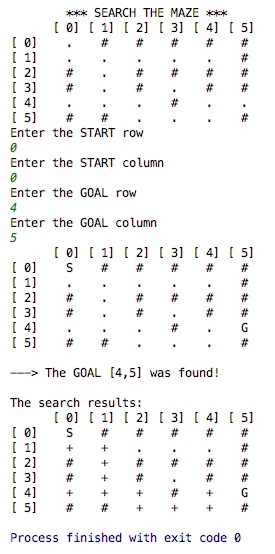


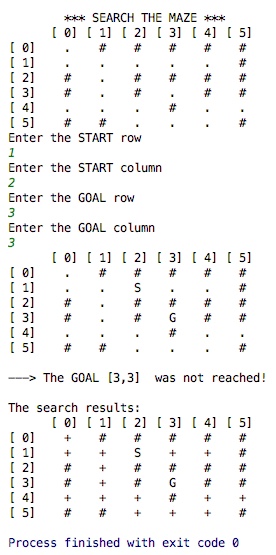


## MazeSolverWithStack

Use stack instead of recursion to implement the previous algorithm, use MazeSolverWithStack.java as the starting point. Your program should produce similar results to the MazeSolver.java. See segment “Using a Stack Instead of Recursion” on page 224 of your textbook.

Sample runs:





## MinInArray

Write a recursive method that returns the smallest integer in an array of integers.

If you divide the array into two pieces - halves, for example - and find the smallest integer in each of the pieces, the smallest integer in the entire array will be the smaller of these two integers. Since you will be searching a portion of the array - for example, the elements array[first] through array[last] - it will be convenient for your method to have three parameters: the array and two indices: first and last.

NOTE: You can refer to the method displayArray in Segment 7.18 in the textbook for the inspiration.

Test your methods with the driver provided in main.