Computer Project

(2017-2019)

Satvik Saha

Class: XI B

Roll number: 24

"Writing code a computer can understand is science. Writing code other programmers can understand is an art."

— **Jason Gorman**

"I am rarely happier than when spending an entire day programming my computer to perform automatically a task that would otherwise take me a good ten seconds to do by hand."

— **Douglas Adams**

Problem 1 An n digit integer $(a_1a_2 \ldots a_n)$, where each digit $a_i \in \{0, 1, \ldots, 9\}$, is said to have *unique digits* if no digits are repeated, i.e., there is no i, j such that $a_i = a_j$ $(i \neq j)$.

Verify whether an inputted number has *unique digits*.

Solution The problem involves simply counting the number of occurrences of each digit in the given number and checking whether any of them exceed 1.

`main (number:Integer)`

1. Initialize an integer array `digits` of length `10`, indexed with integers from `[0]` to `[9]` with all elements set to `0`.
2. If `number` exceeds `0`, proceed. Otherwise, jump to (3).
   (a) Store the last digit[1] of `number` in a temporary variable `d`.
   (b) Increment the integer at the `d` index of `digits`.
   (c) If `digits[d]` exceeds `1`, the number does not have *unique digits*. Display a suitable message, and **exit**.
   (d) Discard the last digit of `number` by performing an integer division by `10` and storing the result back in `number`.
   (e) Jump to (2).
3. The number has *unique digits*. Display a suitable message.
4. **Exit**

[1]The last digit of an integer n is simply n mod 10

Source Code

```java
public class Unique {
	public static void main (String[] args) {
		try {
			/* Parse the first command line argument as the number
			   to check for unique digits */
			long number = Long.parseLong(args[0]);
			if (isUnique(number)) {
				System.out.println("Unique Number!");
			} else {
				System.out.println("Not a Unique Number!");
			}
		} catch (NumberFormatException | IndexOutOfBoundsException e) {
			/* Handle missing or incorrectly formatted arguments */
			System.out.println("Enter 1 argument (number[integer])!");
		}
	}

	public static boolean isUnique (long number) {
		/* Keep track of the number of occurrences of each digit */
		int[] count = new int[10];
		for (long n = Math.abs(number); n > 0; n /= 10) {
			/* Extract the last digit of the number */
			int digit = (int) n % 10;
			count[digit]++;
			if (count[digit] > 1){
				return false;
			}
		}
		return true;
	}
}
```

Variable Description

`Unique::main(String[])`		
`long`	`number`	The inputted number

`Unique::isUnique(long)`		
`long`	`number`	The number to check for uniqueness
`int[]`	`count`	The number of occurrences of each digit
`long`	`n`	Counter, temporarily stores the value of `number`
`int`	`digit`	The last digit in `n`

"Elegance is not a dispensable luxury but a factor that decides between success and failure."

— **Edsger W. Dijkstra**

Problem 2 A *partition* of a positive integer n is defined as a collection of other positive integers such that their sum is equal to n. Thus, if $(a_1, a_2, \ldots, a_k)$ is a partition of n,

$$n = a_1 + a_2 + \cdots + a_k \qquad (a_i \in \mathbb{Z}^+)$$

Display every *unique partition* of an inputted number.

Solution This problem can be solved elegantly using *recursion*[2]. Note that when partitioning a number n, we can calculate the partitions of $(n-1)$ and append 1 to each solution. Similarly, we can append 2 to partitions of $(n-2)$, 3 to partitions of $(n-3)$, and so on. By continuing in this fashion, all cases will be reduced to the single *base case*[3] of finding the partitions of 0, of which there are trivially none.[citation needed]

There is a slight flaw in this algorithm — partitions are often repeated. This can be overcome by imposing the restriction that each new term has to be of a lesser magnitude than the previous. In this way, repeated partitions will be automatically discarded.

`main (target:Integer)`

1. Call `partition(target, target, "")`.
2. **Exit**

`partition (target:Integer, previousTerm:Integer, suffix:String)`

1. If `target` is `0`, display `suffix` and **return**.
2. Initialize a counter `i` to `1`.
3. If `i` is less than or equal to both the `target` and `previousTerm`, proceed. Otherwise, jump to (4).
   (a) Call `partition(target - i, i, suffix + " " + i)`.
   (b) Increment `i` by `1`.
   (c) Jump to (3).
4. **Return**

[2]Recursion occurs when a thing is defined in terms of itself or of its type.

[3]A base case is a case for which the answer is known and can be expressed without recursion.

Source Code

```
1
2  public class Partition {
3         public static void main (String[] args) {
4                 try {
5                         /* Parse the first command line argument as the target sum */
6                         int target = Integer.parseInt(args[0]);
7                         if (target < 1) {
8                                 throw new NumberFormatException();
9                         }
10                        partition(target);
11                } catch (NumberFormatException | IndexOutOfBoundsException e) {
12                        /* Handle missing or incorrectly formatted arguments */
13                        System.out.println("Enter 1 argument (number[natural
                              number])!");
14                }
15        }
16
17        /* Wrapper method for displaying partitions of a number */
18        public static void partition (int target) {
19                partition(target, target, "");
20        }
21
22        /* Display the partitions of the target */
23        public static void partition (int target, int previousTerm, String suffix) {
24                /* Base case : '0' has no partitions */
25                if (target == 0)
26                        System.out.println(suffix);
27                /* Recursively solve for partitions by diminishing the target,
28                   adding that difference to the solution, and partitioning the
29                   remaining sum */
30                for (int i = 1; i <= target && i <= previousTerm; i++)
31                        partition(target - i, i, suffix + " " + i);
32        }
33 }
```

Variable Description

Partition::main(String[])		
int	target	The inputted number

Partition::partition(int)		
int	target	The number to be partitioned

Partition::partition(int, int, String)		
int	target	The number to be partitioned
int	previousTerm	The previous term in the partition sequence
String	suffix	Terms in the sequence calculated so far
int	i	Counter variable, stores the next term in the sequence

"Simplicity is the ultimate sophistication."

— **Leonardo da Vinci**

Problem 3 A *Caesar cipher* is a type of monoalphabetic substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. The positions are circular, i.e., after reaching Z, the position wraps around to A. For example, following is some encrypted text, using a right shift of 5.

```
Plain:   ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher:  FGHIJKLMNOPQRSTUVWXYZABCDE
```

Thus, after mapping the alphabet according to the scheme $A \mapsto 0, B \mapsto 1, \ldots, Z \mapsto 23$, we can define an encryption function E_n, in which a letter x is shifted rightwards by n as follows.

$$E_n(x) \;=\; (x + n) \mod 26$$

The corresponding decryption function D_n is simply

$$D_n(x) \;=\; (x - n) \mod 26$$

Implement a simple version of a *Caesar cipher*, encrypting capitalized plaintext by shifting it by a given value. Interpret positive shifts as rightwards, negative as leftwards.

Solution This problem can be solved simply by exploiting the fact that Unicode characters are already arranged in order, with successive alphabets encoded by consecutive numbers. In addition, the encryption function can be defined exactly as given in the question — characters can be converted to their corresponding codes, manipulated by addition of the `shift`, and converted back into alphabetic form.

`main (shift:Integer, plainText:String)`

1. Normalize `plainText` to uppercase.
2. Normalize `shift` by replacing it with `shift` mod 26.
3. Initialize an empty String `cipherText`.
4. Initialize a counter `i` to `0`.
5. If `i` is less than the length of `plainText`, proceed. Otherwise, jump to (6).
   (a) Store the character in `plainText` at position `i` in a variable `plain`.
   (b) Initialize an empty character `crypt`.
   (c) If `plain` is not an alphabet, assign `plain` to `crypt` and jump to (5g).
   (d) Convert `plain` into a number, such that `A` is mapped to `0`, `B` to `1` and so on. Store this in a temporary variable `n`.

(e) Add `shift` to `n`, calculate its least residue modulo 26[4], and store the result in `n`.
(f) Convert `n` back into a character and store the result in `crypt`.
(g) Append `crypt` to `cipherText`.
(h) Increment `i` by `1` and jump to (5).

6. Display `cipherText`.
7. **Exit**

Source Code

```
1  public class CaesarShift {
2  	public static void main (String[] args) {
3  		try {
4  			/* Parse the first command line argument as the shift */
5  			int shift = Integer.parseInt(args[0]) % 26;
6  			/* Parse the second command line argument as the text to
   			    encrypt */
7  			String plaintext = args[1].toUpperCase();
8  			String ciphertext = "";
9  			for (int i = 0; i < plaintext.length(); i++) {
10 				char plain = plaintext.charAt(i);
11 				char crypt = ' ';
12 				if ('A' <= plain && plain <= 'Z') {
13 					/* Only shift letters of the alphabet */
14 					crypt = numToChar(charToNum(plain) + shift);
15 				} else {
16 					/* Keep special characters intact */
17 					crypt = plain;
18 				}
19 				/* Append the encrypted character to the cipherText */
20 				ciphertext += crypt;
21 			}
22 			System.out.println(ciphertext);
23 		} catch (NumberFormatException | IndexOutOfBoundsException e) {
24 			/* Handle missing or incorrectly formatted arguments */
25 			System.out.println("Enter 2 arguments (shift[integer],
   			    plaintext[text])!");
26 		}
27 	}
28
29 	/* Map letters to numbers */
30 	public static int charToNum (char letter) {
```

[4]The set of integers $K = \{0, 1, 2, \ldots, n-1\}$ is called the least residue system modulo n. The number k such that $k \in K$ and $a \equiv k \pmod{n}$ is called the least residue of a modulo n.

```
31                return Character.toUpperCase(letter) - 'A';
32        }
33
34        /* Map numbers to letters */
35        public static char numToChar (int number) {
36                return (char) ('A' + Math.floorMod(number, 26));
37        }
38  }
```

Variable Description

CaesarShift::main(String[])		
int	shift	The inputted 'shift'
String	plainText	The text to encrypt
String	cipherText	The encrypted text
int	i	Counter variable, stores the position in plainText
char	plain	The character to encrypt
char	crypt	The encrypted form of plain
CaesarShift::charToNum(char)		
char	letter	The character to convert to an integer
CaesarShift::numToChar(int)		
int	number	The number to convert to a character

"There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-1 errors."

— Leon Bambrick

Problem 4 A *palindrome* is a sequence of characters which reads the same backwards as well as forwards. For example, `madam`, `racecar` and `kayak` are words which are palindromes. Similarly, the sentence "`A man, a plan, a canal -- Panama!`" is also a palindrome.

Analyze a sentence of input and display all *words* which are palindromes. If the entire *sentence* is also a palindrome, display it as well.

(A word is an unbroken sequence of characters, separated from other words by whitespace. Ignore single letter words such as I *and* a*. Ignore punctuation, numeric digits, whitespace and case while analyzing the entire sentence.)*

Solution The main challenge here is intelligently dividing a *sentence* into its component *words*. Verifying whether a sequence of characters is a palindrome is fairly simple — extracting those characters from a string of alphabets, numbers, punctuation and whitespace is not.

The main idea behind isolating words from sentences is to define two *markers* — a `start` to keep track of the boundary between whitespace and letters, and an `end` to mark the boundary between letters and whitespace. In this way, the markers can inch their way along the sentence, isolating words in the process. Managing the order of condition checking and incrementing of counters does require some careful manoeuvring in order to avoid any *off-by-1 errors*[5] — any of which would inevitably result in incorrect, hence undesirable output.[citation needed]

`main ()`

1. Accept a string as input, store it in a variable `sentence`.
2. Call `checkWords(sentence)` and `checkSentence(sentence)`. Store the returned values in booleans.
   (a) If either of them is `true`, set a boolean `foundPalindrome` to `true`, otherwise set it to `false`.
3. Display a suitable message if `foundPalindrome` is `false`.
4. **Exit**

[5] An off-by-one error often occurs in computer programming when an iterative loop iterates one time too many or too few.

`checkWords (sentence:String)`

1. Initialize a boolean `foundPalindrome` to false.
2. Initialize two integer counters: `start` to `-1`, `end` to `0`.
3. If `end` is less than the length of `sentence`, proceed. Otherwise, jump to (4).
   (a) Increment `start` as long as the character at the `[start + 1]` position in `sentence` is whitespace.
   (b) Assign `end` to `start`.
   (c) Increment `end` as long as it does not exceed the length of `sentence` and the character at the `[end]` position in `sentence` is not whitespace.
   (d) Assign the string of characters between `start` and `end` from `sentence` (inclusive, exclusive) to a variable `word`.
   (e) Call `isPalindrome(word)`. If `word` is a palindrome:
      i. Set `foundPalindrome` to `true`.
      ii. Display `word`.
   (f) Assign `end - 1` to `start`.
   (g) Jump to (3)
4. **Return** `foundPalindrome`

`checkSentence (sentence:String)`

1. Call `isPalindrome(sentence)`. If `sentence` is a palindrome:
   (a) Display `word`.
   (b) **Return** `true`.
2. **Return** `false`.

`isPalindrome (text:String)`

1. Normalize `text` by converting it into uppercase and removing all non-alphabetic characters.
2. Let the length of `text` be labeled temporarily as `t`.
3. Initialize two integer counters: `i` to `0`, `j` to `l - 1`.
4. If `i` is less than `j`, proceed. Otherwise, jump to (5).
   (a) If the characters at positions `i` and `j` in `text` are not equal, **return** `false`.
   (b) Increment `i` by `1`.
   (c) Decrement `j` by `1`.
   (d) Jump to (4)
5. **Return** `true` only if `text` is longer than one character. Otherwise, **return** `false`.

Source Code

```
import java.util.Scanner;

public class Palindrome {
	public static void main (String[] args) {
		System.out.print("Enter your sentence : ");
		String sentence = (new Scanner(System.in)).nextLine().trim();
		/* Keep track of whether palindromes have been found */
		boolean foundPalindrome = false;
		System.out.println("Palindromes : ");
		foundPalindrome |= checkWords(sentence);
		foundPalindrome |= checkSentence(sentence);
		if (!foundPalindrome) {
			System.out.println("(No palindromes found!)");
		}
	}

	/* Slice a sentence into words and check each individually */
	public static boolean checkWords (String sentence) {
		boolean foundPalindrome = false;
		int start = -1;
		int end = 0;
		while (end < sentence.length()) {
			while (Character.isWhitespace(sentence.charAt(++start)));
			end = start;
			while (end < sentence.length() &&
			    !Character.isWhitespace(sentence.charAt(end++)));
			String word = sentence.substring(start, end).trim();
			if (isPalindrome(word)) {
				foundPalindrome = true;
				System.out.println(getAlphabets(word));
			}
			start = end - 1;
		}
		return foundPalindrome;
	}

	/* Check the sentence as a whole */
	public static boolean checkSentence (String sentence) {
		if (isPalindrome(sentence)) {
			System.out.println("The sentence '" + sentence + "' is a
			    palindrome.");
			return true;
		}
		return false;
```

```
43          }
44
45          /* Check whether a piece of text is identical forward as well as backwards */
46          public static boolean isPalindrome (String text) {
47                  String rawText = getAlphabets(text).toUpperCase();
48                  for (int i = 0, j = rawText.length() - 1; i < j; i++, j--) {
49                          if (rawText.charAt(i) != rawText.charAt(j)) {
50                                  return false;
51                          }
52                  }
53                  /* Make sure that the text is not just one letter */
54                  return (rawText.length() > 1);
55          }
56
57          /* Strip a piece of text of all characters except alphabetic ones */
58          public static String getAlphabets (String text) {
59                  String rawText = "";
60                  for (int i = 0; i < text.length(); i++) {
61                          if (Character.isAlphabetic(text.charAt(i))) {
62                                  rawText += text.charAt(i);
63                          }
64                  }
65                  return rawText;
66          }
67  }
```

Variable Description

`Palindrome::main(String[])`		
`String`	`sentence`	Stores the text to check for palindromes
`boolean`	`foundPalindrome`	Stores whether palindromes have been found

`Palindrome::checkWords(String)`		
`String`	`sentence`	Stores the sentence to divide into words
`boolean`	`foundPalindrome`	Stores whether palindromes have been found
`int`	`start`	Counter variable, stores the index of the start of a word
`int`	`end`	Counter variable, stores the index of the end of a word
`String`	`word`	Stores words in `sentence`, extracted between `start` and `end`

`Palindrome::checkSentence(String)`		
`String`	`sentence`	Stores the sentence to divide into words

`Palindrome::isPalindrome(String)`		
`String`	`text`	Stores the text to check
`String`	`rawText`	Stores only alphabets from `text`
`int`	`i`	Counter variable, stores the current index in `text`

`Palindrome::getAlphabets(String)`		
`String`	`text`	Stores the text to extract alphabets from
`String`	`rawText`	Stores only alphabets from `text`
`int`	`i`	Counter variable, stores the current index in `text`

"In programming the hard part isn't solving problems, but deciding what problems to solve."

— **Paul Graham**

Problem 5 A *prime number* (or a *prime*) is a natural number greater than 1 that has no positive divisors other than 1 and itself.

Display all primes upto a given limit, along with their number.

Solution This problem can be tackled in a multitude of ways.[citation needed] We could define a function for checking the primality of a given number, then iterate through all numbers in the required range. A common way of checking for primality is *trial division*. It consists of testing whether the number n is a multiple of any integer between 2 and $\sqrt{n}$. Although this works well enough for small numbers, repeating this consecutively for very large inputs is tedious and inefficient. Since the problem consists of identifying primes in a *range*, and not individually, we can make use of more efficient methods.

The *Sieve of Eratosthenes* is a simple, ancient algorithm for finding all prime numbers up to any given limit. It does so by iteratively marking as composite the multiples of each prime, starting with the first prime number, 2. As a result, when a prime p is found, none of its multiples will be tested further for primality — they are eliminated early on. In comparison, *trial division* has worse theoretical complexity than that of the *Sieve of Eratosthenes* in generating ranges of primes. When testing each prime, the optimal trial division algorithm uses all prime numbers not exceeding its square root, whereas the Sieve of Eratosthenes produces each composite only from its prime factors.

`main (upperLimit:Integer)`

1. Create a new `SieveOfEratosthenes`, pass it `upperLimit` and assign it to `sieve`.
2. Call `sieve->sievePrimes()`.
3. Display the indices which correspond to `true` in the boolean array `sieve->primes`.
4. **Exit**

`SieveOfEratosthenes (upperLimit:Integer)`

1. Initialize a boolean array `primes`, indexed with integers from `[0]` to `[upperLimit - 1]`, with all elements set to `true`.
2. Set `primes[0]` and `primes[1]` to `true`.
3. **Define** the function `SieveOfEratosthenes::sievePrimes()` and **return** the resultant object.

`SieveOfEratosthenes::sievePrimes ()`

1. Initialize an integer variable `prime` to 2.
2. If `prime` is less than the square root of `upperLimit`, proceed. Otherwise, **return**.
   (a) Initialize an integer variable `multiple` to the square of `prime`.
   (b) If `multiple` is less than `upperLimit`, proceed. Otherwise, jump to (2c).
      i. Set `primes[multiple]` to `false`.
      ii. Increment `multiple` by `prime`.
      iii. Jump to (2b)
   (c) Increment `prime` until `primes[prime]` is `true`.
   (d) Jump to (2).
3. **Return**

Source Code

```
1  public class SieveOfEratosthenes {
2        private final int upperLimit;
3        private boolean[] primes;
4
5        /* Initialize the list of numbers using an uper limit */
6        public SieveOfEratosthenes (int upperLimit) {
7              this.upperLimit = upperLimit;
8              this.initPrimes();
9        }
10
11       public boolean[] getPrimes () {
12             return primes;
13       }
14
15       /* Initialize all value to 'prime' by default */
16       public void initPrimes () {
17             this.primes = new boolean[upperLimit];
18             /* Mark known values as 'not prime' */
19             primes[0] = false;
20             primes[1] = false;
21             for (int i = 2; i < upperLimit; i++)
22                   primes[i] = true;
23       }
24
25       /* Iteratively sieve the numbers to leave primes behind */
26       public void sievePrimes () {
27             /* Start with the first prime */
28             int prime = 2;
```

```
                while ((prime * prime) < upperLimit) {
                        /* Start with the first multiple not crossed off */
                        int multiple = prime * prime;
                        while (multiple < upperLimit) {
                                /* Cross multiples of a prime off the list */
                                primes[multiple] = false;
                                multiple += prime;
                        }
                        /* Skip forward to the next prime */
                        while (!primes[++prime]);
                }
        }
}
```

```
public class Primes {
        public static void main (String[] args) {
                try {
                        /* Parse the first command line argument as the upper limit
                           on primes to calculate */
                        int upperLimit = Integer.parseInt(args[0]);
                        if (upperLimit < 2) {
                                throw new NumberFormatException();
                        }
                        SieveOfEratosthenes sieve = new
                            SieveOfEratosthenes(upperLimit);
                        sieve.sievePrimes();
                        showPrimes(sieve.getPrimes());
                } catch (NumberFormatException | IndexOutOfBoundsException e) {
                        /* Handle missing or incorrectly formatted arguments */
                        System.out.println("Enter 1 argument (limit[integer, >1])!");
                        System.out.println("(Primes will be dislayed up to, not
                            including 'limit')");
                }
        }

        /* Display all primes calculated */
        public static void showPrimes (boolean[] primes) {
                int primeCount = 0;
                /* Format all number to the same width */
                int maxLength = Integer.toString(primes.length).length();
                for (int i = 0; i < primes.length; i++) {
                        /* If 'i' is prime, primes[i] will be marked 'true' */
                        if (primes[i]) {
                                System.out.printf("%" + maxLength + "d ", i);
                                primeCount++;
```

```
30                      }
31                  }
32                  System.out.println("\nTotal number of primes : " + primeCount);
33          }
34  }
```

Variable Description

SieveOfEratosthenes		
int	upperLimit	The number of integers to sieve
boolean[]	primes	Primes, with contents indicating the primality of the index

SieveOfEratosthenes::initPrimes()		
int	i	Counter variable

SieveOfEratosthenes::sievePrimes()		
int	prime	Counter variable, stores current primes found
int	multiple	Counter variable, stores the multiples of prime

Primes::main(String[])		
int	upperLimit	The highest integer to check for primality (exclusive)
SieveOf Eratosthenes	sieve	An object capable of sieving primes

Primes::showPrimes(boolean[])		
boolean[]	primes	Primes, with contents indicating the primality of the index
int	primeCount	The number of primes found
int	maxLength	The length of the longest number to display
int	i	Counter variable, stores the current integer to check for primality

"Any fool can use a computer. Many do."

— Ted Nelson

Problem 6 Design a simple interface for an examiner which can format and display marks scored by a group of students in a particular examination. Calculate the percentage scored by each candidate and display the list of students and percentages in an ASCII bar chart, arranged alphabetically.

Solution This problem calls for a fairly straightforward flow of logic. The main goal is to present the user with a simple way of providing input, along with nicely formatted output.

`main (upperLimit:Integer)`

1. Input the maximum marks allotted for the examination as a floating point. Store it as `maxMarks`.
2. Input the total number of students whose marks are to be recorded as an integer. Store it as `numberOfStudents`.
3. Create a new `Marksheet`, pass it `maxMarks`, `numberOfStudents` and assign it to `sheet`.
4. Initialize an integer counter `i` to `0`;
5. If `i` is less than `numberOfStudents`, proceed. Otherwise, jump to (6).
   (a) Input a student's name as a string. Store it as `name`.
   (b) Input the student's marks as a floating point. Store it as `marks`.
   (c) Call `sheet->addMarks(name, marks)`.
   (d) Jump to (5).
6. Call `sheet->sortByName()`.
7. Call `sheet->displayChart()`.
8. Call `sheet->sortMaxScorers()`.
9. **Exit**

`Marksheet (maxMarks:FloatingPoint, numberOfStudents:Integer)`

1. Initialize a string array `names`, indexed with integers from `[0]` to `[numberOfStudents - 1]`.
2. Initialize a floating point array `marks`, indexed with integers from `[0]` to `[numberOfStudents - 1]`.
3. Initialize an integer counter `lastStudent` to `-1`.
4. **Define** the functions:

(a) `Marksheet::addMarks(name, score)`
(b) `Marksheet::sortByName()`
(c) `Marksheet::displayChart()`
(d) `Marksheet::displayMaxScorers()`

5. **Return** the resultant object.

`Marksheet::addMarks (name:String, score:FloatingPoint)`

1. Increment `lastStudent` by `1`.
2. Set the `names[lastStudent]` to `name`.
3. Set the `marks[lastStudent]` to `score`.
4. **Return**

`Marksheet::sortByName ()`

1. Assign `lastStudent` to `right`.
2. If `right` exceeds `0`, proceed. Otherwise, **return**.
   (a) Initialize an integer counter `i` to `1`.
   (b) If `i` is less than or equal to `right`, proceed. Otherwise, jump to (2c).
      i. If `names[i-1]` comes lexicographically after `names[i]`:
         A. Swap the elements at `names[i-1]` and `names[i]`.
         B. Swap the elements at `marks[i-1]` and `marks[i]`.
      ii. Jump to (2b).
   (c) Jump to (2).

`Marksheet::displayChart ()`

1. For every string `name` in `names`:
   (a) Calculate the length of the bar in the chart as a fraction of the screen width. Store the calculated number of characters to display as `points`.
   (b) Display `name`, a string of suitable characters for the bar of length `points`, along with the percentage scored.
2. **Return**

`Marksheet::displayMaxScorers ()`

1. Calculate the maximum floating point in `marks` and store it as `maxScore`.
2. For every integer `i` between `0` and `numberOfStudents` (inclusive, exclusive) such that `marks[i]` is equal to the `maxScore`, display `names[i]`.
3. **Return**

Source Code

```
public class Marksheet {
	public static final int SCREEN_WIDTH = 100;
	private final double maxMarks;
	private final int numberOfStudents;
	private int lastStudent;
	private String[] names;
	private double[] marks;

	/* Initialize some final data */
	public Marksheet (double maxMarks, int numberOfStudents) {
		this.maxMarks = maxMarks;
		this.numberOfStudents = numberOfStudents;
		this.names = new String[numberOfStudents];
		this.marks = new double[numberOfStudents];
		this.lastStudent = -1;
	}

	/* Add names and marks to the stack */
	public boolean addMarks (String name, double score) {
		try {
			names[++lastStudent] = name;
			marks[lastStudent] = score;
			return true;
		} catch (IndexOutOfBoundsException e) {
			return false;
		}
	}

	/* Display the names and percentages in a bar chart */
	public void displayChart () {
		System.out.println(Marksheet.multiplyString("-",
		    Marksheet.SCREEN_WIDTH));
		for (int i = 0; i <= lastStudent; i++) {
			/* Calculate the fraction of marks earned */
			double fraction = marks[i] / maxMarks;
			String name = (names[i].length() < 16)
					? names[i]
					: (names[i].substring(0,13) + "...");
			int points = (int) (fraction * (SCREEN_WIDTH - 34));
			/* Generate and pad the bar to display */
			String bar = multiplyString("*", points)
				+ multiplyString(" ", SCREEN_WIDTH - 34 - points);
			System.out.printf("| %16s | %s | %6.2f %% |%n"
									, name
```

```
                                                , bar
                                                , fraction * 100);
        }
        System.out.println(Marksheet.multiplyString("-",
            Marksheet.SCREEN_WIDTH));
    }

    /* Display the name of students with the highest score */
    public void displayMaxScorers () {
        String maxScorers = "";
        double maxScore = getMaxScore();
        for (int i = 0; i <= lastStudent; i++) {
            if (marks[i] == maxScore) {
                maxScorers += ", " + names[i];
            }
        }
        System.out.println(maxScorers.substring(1)
                           + " scored the highest ("
                           + maxScore + "/"
                           + maxMarks + ")");
    }

    /* Sort the names and associated marks lexicographically */
    public void sortByName () {
        for (int right = lastStudent; right > 0; right--)
            for (int i = 1; i <= right; i++)
                if (names[i-1].compareToIgnoreCase(names[i]) > 0)
                    swapRecords(i, i - 1);

    }

    /* Get the value of the highest score */
    public double getMaxScore () {
        double max = Integer.MIN_VALUE;
        for (int i = 0; i <= lastStudent; i++) {
            max = Math.max(max, marks[i]);
        }
        return max;
    }

    /* Utility function to swap student records */
    private void swapRecords (int x, int y) {
        String tempName = names[x];
        double tempMark = marks[x];
        names[x] = names[y];
        marks[x] = marks[y];
```

```
89                names[y] = tempName;
90                marks[y] = tempMark;
91        }
92
93        /* Utility funtion for repeating strings */
94        public static String multiplyString (String s, int n) {
95                String out = "";
96                while (n --> 0)
97                        out += s;
98                return out;
99        }
100 }
```

```
1   import java.util.Scanner;
2   import java.util.InputMismatchException;
3
4   public class ScoreRecorder {
5         public static void main (String[] args) {
6                 /* Create an object capable of maning input */
7                 Scanner inp = new Scanner(System.in);
8                 double maxMarks = 0.0;
9                 int numberOfStudents = 0;
10                try {
11                        System.out.print("Enter the maximum marks allotted for each
                              student : ");
12                        maxMarks = inp.nextDouble();
13                        System.out.print("Enter the total number of students : ");
14                        numberOfStudents = inp.nextInt();
15                        /* Check for any erraneous data */
16                        if (maxMarks <= 0) {
17                              System.out.println("Maximum marks must be positive!");
18                              System.exit(0);
19                        }
20                        if (numberOfStudents <= 0) {
21                              System.out.println("Number of students must be
                                    positive!");
22                              System.exit(0);
23                        }
24                        /* Create an object capable of recording scoresheets */
25                        Marksheet sheet = new Marksheet(maxMarks, numberOfStudents);
26                        System.out.println("Enter " + numberOfStudents + " students'
                              names and marks : ");
27                        /* Accept student data */
28                        for (int i = 0; i < numberOfStudents; i++) {
29                              String name = "";
```

```
30                              while (!inp.hasNextDouble()) {
31                                      name += inp.next() + " ";
32                              }
33                              double marks = inp.nextDouble();
34                              if (marks <= 0 || marks > maxMarks) {
35                                      System.out.println("Marks must be within 0.0 and
                                            " + maxMarks + "!");
36                                      System.exit(0);
37                              }
38                              sheet.addMarks(name.trim(), marks);
39                      }
40                      /* Sort and display */
41                      sheet.sortByName();
42                      sheet.displayChart();
43                      sheet.displayMaxScorers();
44              } catch (InputMismatchException e) {
45                      /* Handle missing or incorrectly formatted arguments */
46                      System.out.println("Invalid Input!");
47                      System.exit(0);
48              }
49      }
50  }
```

Variable Description

Marksheet		
int	SCREEN_WIDTH	Number of characters to use in the display width
double	maxMarks	The maximum marks allotted for the examination
int	numberOf Students	The number of students whose marks are to be recorded
int	lastStudent	The index number of the last student added to the marksheet
String[]	names	The names of the students
double[]	marks	The marks of the students

Marksheet::addMarks(String, double)		
String	name	The name of the student to be added
double	score	The marks of the student to be added

Marksheet::displayChart()		
int	i	Counter variable
double	fraction	The fraction on marks scored over the maximum marks

String	name	Temporarily stores a formatted version of a student's name
int	points	The number of characters to display in the bar chart
String	bar	The bar in the chart, along with whitespace padding

Marksheet::displayMaxScorers()		
String	maxScorers	The list of highest scoring students
double	maxScore	The highest score
int	i	Counter variable

Marksheet::sortByName()		
int	right	Counter variable
int	i	Counter variable

Marksheet::getMaxScore()		
double	max	The maximum score in marks
int	i	Counter variable

Marksheet::swapRecords(int, int)		
int	x, y	The indices of the records to swap
String	tempName	Temporary storage of a name
double	tempMark	Temporary storage of a mark

Marksheet::multiplyString(String, int)		
String	s	The string to multiply
int	n	The number of times to multiply s
String	out	The string containing n copies of s

ScoreRecorder::main(String[])		
Scanner	inp	The input managing object
double	maxMarks	The maximum marks allotted for the examination
int	numberOf Students	The number of students whose marks are to be recorded
Marksheet	sheet	An object capable of managing student records
int	i	Counter variable
String	name	The name of the student to be added
double	marks	The marks of the student to be added

"To iterate is human, to recurse divine"

— L. Peter Deutsch

Problem 7 The *determinant* of a square matrix $A_{n,n}$ is defined recursively as follows.

$$det(A_{n,n}) = \begin{vmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{vmatrix} = \sum_{j=1}^{n} (-1)^{i+j} a_{i,j} \cdot det(M_{i,j})$$

where $M_{i,j}$ is defined as the minor of $A_{n,n}$, an $(n-1) \times (n-1)$ matrix formed by removing the ith row and jth column from $A_{n,n}$.

The determinant of a (2×2) matrix is simply given by

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

For example, the determinant of a (3×3) matrix is given by the following expression.

$$\begin{aligned} \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} &= a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} \\ &= aei + bfg + cdh - ceg - bdi - afh \end{aligned}$$

Calculate the *determinant* of an inputted $(n \times n)$ square matrix.

Solution This problem offers the opportunity to showcase the power of recursive functions. Here, the complex task of calculating the determinant of a large matrix can be subdivided into multiple smaller tasks. In fact, each of these tasks is precisely the same as the larger one — the only difference is the size of the matrices. Eventually, the problem reduces to finding the determinants of multiple (2×2) matrices. The values thus obtained can be pieced together to form the final answer.

`main ()`

1. Input the size (number of rows/columns) of the square matrix. Store it as `size`.
2. Create a new `SquareMatrix`, pass it `size`, and assign it to `matrix`.
3. For each `i` $\in \{1, 2, \ldots,$ `size`$\}$:

(a) For each `j` $\in \{1, 2, \ldots,$ `size`$\}$:
   i. Input an integer as `n`.
   ii. Set the element at `[i, j]` of `matrix` to `n`.

4. Call `matrix->getDeterminant()` and display the returned value.
5. **Exit**

`Matrix (rows:Integer, columns:Integer)`

1. Initialize an integer array of integer arrays `elements`, indexed with integers from `[1]` to `[rows]`, with each contained integer array indexed with integers from `[1]` to `[columns]` .
2. **Return** the resultant object.

`SquareMatrix (size:Integer)`

1. **Define** the functions:
   (a) `SquareMatrix::getDeterminant()`
   (b) `SquareMatrix::getMinorMatrix(row, column)`
2. **Return** a `Matrix`, with both `rows` and `columns` set to `size`.

`SquareMatrix::getDeterminant ()`

1. If the `size` is `1`, **return** the only element (`elements[1, 1]`).
2. If the `size` is `2`, **return** (`elements[1, 1]`$\times$`elements[2, 2]`) $-$ (`elements[1, 2]`$\times$`elements[2, 1]`).
3. Initialize an integer variable `determinant` to `0`.
4. For each `i` $\in \{1, 2, \ldots,$ `size`$\}$:
   (a) Call `this->getMinorMatrix(i, i)->getDeterminant()`. Store the result in `d`.
   (b) Add $((-1)^{\texttt{i}+1} \times$ `matrix[1, i]` $\times$ `d`) to `determinant`.
5. **Return** `determinant`.

`SquareMatrix::getMinorMatrix (row:Integer, column:Integer)`

1. Create a new `SquareMatrix`, pass it `(size - 1)`, and assign it to `minor`.
2. Copy all elements from `this` to `minor`, except for those at position `[row, *]` or `[*, column]`.
3. **Return** `minor`.

Source Code

```java
public class Matrix {
	protected final int rows;
	protected final int columns;
	protected int[][] elements;

	/* Initialize a matrix of a given order */
	public Matrix (int rows, int columns) {
		this.rows = rows;
		this.columns = columns;
		this.elements = new int[rows][columns];
	}

	public int getRows () {
		return this.rows;
	}

	public int getColumns () {
		return this.columns;
	}

	/* Set elements in the matrix using natural indices */
	public void setElementAt (int element, int row, int column) {
		if (row < 1 || row > rows || column < 1 || column > columns)
			return;
		elements[row-1][column-1] = element;
	}

	/* Get elements from the matrix using natural indices */
	public int getElementAt (int row, int column) {
		if (row < 1 || row > rows || column < 1 || column > columns)
			return Integer.MIN_VALUE;
		return elements[row-1][column-1];
	}
}
```

```java
public class SquareMatrix extends Matrix {
	protected int size;

	/* Initialize the matrix with the same number of rows and columns */
	public SquareMatrix (int size) {
		super(size, size);
		this.size = size;
	}

```

```
10          public int getSize () {
11                  return this.size;
12          }
13
14          /* Recursively calculate the determinant of the matrix */
15          public int getDeterminant () {
16                  /* Base cases */
17                  if (this.size == 1)
18                          return getElementAt(1, 1);
19                  if (this.size == 2)
20                          return (getElementAt(1, 1) * getElementAt(2, 2))
21                                  - (getElementAt(1, 2) * getElementAt(2, 1));
22                  int determinant = 0;
23                  /* Accumulate the determinants of minors with alternating signs */
24                  for (int i = 1; i <= size; i++)
25                          determinant += ((int) Math.pow(-1, 1+i))
26                                          * getElementAt(1, i)
27                                          * getMinorMatrix(1, i).getDeterminant();
28                  return determinant;
29          }
30
31          /* Get the minor matrix by removing a row and a column */
32          public SquareMatrix getMinorMatrix (int row, int column) {
33                  /* Check bounds */
34                  if (row < 1 || row > size || column < 1 || column > size)
35                          return null;
36                  if (this.size <= 1)
37                          return new SquareMatrix(0);
38                  SquareMatrix minor = new SquareMatrix(this.size - 1);
39                  for (int i = 1, p = 1; p < size; i++, p++) {
40                          /* Skip 'row' */
41                          if (i == row)
42                                  i++;
43                          for (int j = 1, q = 1; q < size; j++, q++) {
44                                  /* Skip 'column' */
45                                  if (j == column)
46                                          j++;
47                                  /* Copy values into the new matrix */
48                                  minor.setElementAt(this.getElementAt(i, j), p, q);
49                          }
50                  }
51                  return minor;
52          }
53  }
```

```java
import java.util.Scanner;

public class Determinant {
	public static void main (String[] args) {
		/* Create an object for managing input */
		Scanner inp = new Scanner(System.in);
		try {
			System.out.print("Enter the size of the (size X size) square
			    matrix : ");
			int size = inp.nextInt();
			/* Create a sqare matrix which has suitable methods for
			    calculation */
			SquareMatrix matrix = new SquareMatrix(size);
			System.out.println("Enter " + (size * size) + " integers : ");
			for (int i = 1; i <= size; i++)
				for (int j = 1; j <= size; j++)
					matrix.setElementAt(inp.nextInt(), i, j);
			System.out.println("\nThe determinant is : " +
			    matrix.getDeterminant());
		} catch (Exception e) {
			/* Handle missing or incorrectly formatted arguments */
			System.out.println("Invalid Input!");
		}
	}

	/* Display the matrix in a neat format */
	public static void showMatrix (Matrix m) {
		for (int i = 1; i <= m.getRows(); i++) {
			for (int j = 1; j <= m.getColumns(); j++) {
				System.out.printf("%4d ", m.getElementAt(i, j));
			}
			System.out.println();
		}
	}
}
```

Variable Description

Matrix		
int	rows	Number of rows in the matrix
int	columns	Number of columns in the matrix
int[][]	elements	The array of integer arrays, storing the elements of the matrix

SquareMatrix		
int	size	Number of both rows and columns in the matrix

SquareMatrix::getDeterminant()		
int	determinant	The determinant of the SquareMatrix
int	i	Counter variable

SquareMatrix::getMinorMatrix(int, int)		
int	row	The row to remove from the matrix
int	column	The column to remove from the matrix
SquareMatrix	minor	The matrix obtained by removing row and column
int	i, j	Counter variables

Determinant::main(String[])		
Scanner	inp	The input managing object
int	size	Number of both rows and columns in the matrix
SquareMatrix	matrix	The matrix whose determinant is to be calculated
int	i, j	Counter variables

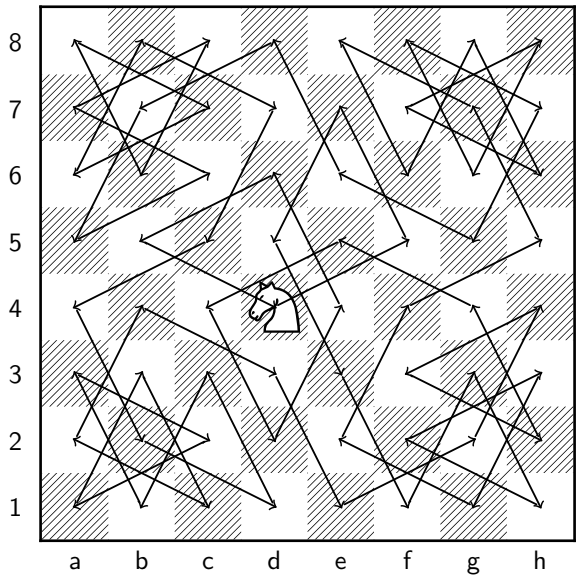
Determinant::showMatrix(Matrix)		
Matrix	m	The matrix to display
int	i, j	Counter variables

"My project is 90% done. I hope the second half goes as well."

— **Scott W. Ambler**

Problem 8 A *Knight's Tour* is a sequence of moves of a knight on a chessboard such that the *knight* visits every square only once. If the knight ends on a square that is one knight's move from the beginning square, the tour is *closed* forming a closed loop, otherwise it is *open*.

There are many ways of constructing such paths on an empty board. On an 8×8 board, there are no less than $26,534,728,821,064$ *directed*[6] *closed* tours. Below is one of them.

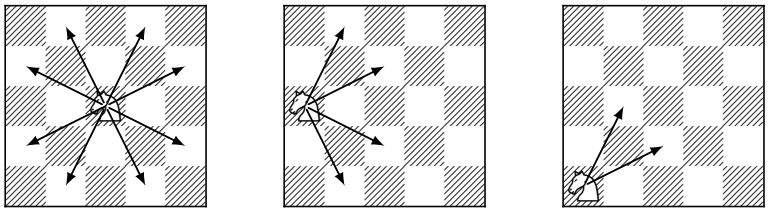


Construct a *Knight's Tour* (*open* or *closed*) on an $n \times n$ board, starting from a given square.

(Mark each square with the move number on which the knight landed on it. Mark the starting square 1*.)*

[6]Two tours along the same path that travel in opposite directions are counted separately, as are rotations and reflections.

Solution A knight on a chessboard can move to a square that is two squares away horizontally and one square vertically, or two squares vertically and one square horizontally.



The mobility of a knight can make varies greatly with its position on the board — near the centre, it can jump to one of 8 squares while when in a corner, it can jump to only 2. On the other hand, the number of possible *sequences* of squares a knight can traverse grows extremely quickly. Although it may seem that a simple *brute force* search can quickly find one of *trillions* of solutions, there are approximately 4×10^{51} different paths to consider on an 8×8 board. For even larger boards, iterating through every possible path is clearly impractical.[citation needed]

This problem calls for implementing a *backtracking*[7] *algorithm*, coupled with some *heuristic*[8] to speed up the search. One such heuristic is *Warnsdorf's Rule*.

> position The knight is moved so that it always proceeds to the square from which the knight will have the *fewest* onward moves.

This allows us to define a ranking algorithm for each possible path — the positions which result in the smallest number of further moves, or is furthest away from the board's centre will be investigated first. In case of a tie, we can either proceed without making any changes to the already existing positions, or introduce a random element. This has the effect of producing different results on successive executions, giving a variety of solutions.

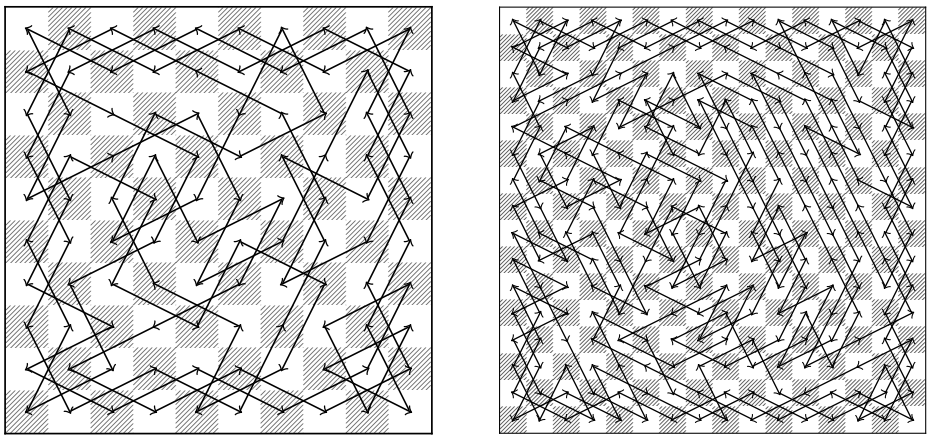
One drawback of resolving ties randomly is that an early "wrong" choice in the position tree can force the calculation of every resulting path without reaching a solution, effectively reducing the algorithm to a brute force search. This is especially problematic

[7] Backtracking is a general algorithm for finding some or all solutions to some computational problems that incrementally builds candidates to the solutions, and abandons each partial candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

[8] A heuristic technique is any approach to problem solving that employs a practical method not guaranteed to be optimal or perfect, but sufficient for the immediate goals. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution.

for large boards, where it may take hours to backtrack and reach a solution. Thus, the "randomness factor" should be adjusted according to the board size.

A high randomness can be useful for searching specifically for *closed tours*, as a randomness of 0 simply produces the same solution every time (which may or may not be closed). Below are some tours generated by the program.



The tendency of the path to remain close to the edges of the board, where the mobility of the knight is restricted, is clearly evident.

`main (boardSize:Integer, initSquare:Position, randomness:FloatingPoint)`

1. Create a new `TourSolver`, pass it `boardSize`, `initSquare`, `randomness`, and assign it to `t`.
2. Call `t->getSolution()`. Store the returned move stack as `solution`.
3. Display the board obtained by calling `t->getBoard()` along with the moves in `solution`.
4. **Exit**

`TourSolver (size:Integer, initSquare:Position, randomness:FloatingPoint)`

1. Initialize an integer arrays of integer arrays indexed with integers from `[1]` to `[size]`, simulating a chessboard. Store it as `board`, which records the move numbers on which the knight lands on it.
2. Initialize a `Position` stack `path`, along with methods to add and remove `Position`'s from it.
3. Set an integer counter `numberOfMoves` to `0`, as part of the `path` stack.
4. **Define** the functions:
   (a) `TourSolver::solve(p)`

   (b) `TourSolver::getPossibleMoves(p)`
5. **Return** the resultant object.

`TourSolver::solve (p:Position)`

1. If the `path` stack is full, **return** `true`, indicating that the tour has been solved.
2. Call `this->getPossibleMoves(p)`. Store the returned list of possible legal moves as `moves`.
3. Sort `moves`, ranking each possible position according to *Warnsdorf's Rule*.
4. For every `move` in the list `moves`:
   (a) Push `move` onto the `path` stack and `board`.
   (b) If the call `this->solve(move)` returns `true`, **return** `true`. Otherwise, pop `move` from the `path` stack and `board` (*backtrack*).
5. If the list `moves` has been exhausted, **return** `false`, indicating that there are no solutions from the position `p` for that particular move stack.

`TourSolver::getPossibleMoves (p:Position)`

1. Initialize a list of moves `possibleMoves`.
2. For every possible square `move` a knight can jump to from `p` (on an empty board):
   (a) If `move` is currently a legal move, without falling outside the board or on a previously traversed square, add it to `possibleMoves`.
3. **Return** `possibleMoves`

Source Code

```
public class TourSolver {
	private final int size;
	private Position[] path;
	private int numberOfMoves;
	private int[][] board;
	private int[][] degreesOfFreedom;
	private Position initPosition;
	private double tieBreakRandomness;

	/* Store the list of possible changes in the 'x' and 'y' coordinates of
	   a knight on an empty board */
	private static final int[][] KNIGHT_MOVES = {
		{-1, -2}, {-1, 2}, {1, -2}, {1, 2},
		{-2, -1}, {-2, 1}, {2, -1}, {2, 1}
	};

	/* Initialize the board and move stack */
	public TourSolver (int size, Position initPosition, double randomness) {
		this.size = size;
		this.initPosition = initPosition;
		this.tieBreakRandomness = randomness / 2.0;
		this.path = new Position[size * size];
		this.numberOfMoves = 0;
		this.initBoard();
		this.initDegreesOfFreedom();
	}

	/* Reset the board */
	public void resetSolution () {
		this.path = new Position[size * size];
		this.numberOfMoves = 0;
		this.initBoard();
	}

	/* Initialize a blank board */
	private void initBoard () {
		board = new int[size][size];
		for (int i = 0; i < size; i++)
			for (int j = 0; j < size; j++)
				board[i][j] = 0;
	}

	/* Calculate the mobility of a knight on each square */
	private void initDegreesOfFreedom () {
```

```
45              degreesOfFreedom = new int[size][size];
46              for (int i = 0; i < size; i++)
47                      for (int j = 0; j < size; j++)
48                              degreesOfFreedom[i][j] = getPossibleMovesCount(new
                                    Position(i, j));
49      }
50
51      /* Push a move onto the move stack, add it to the board */
52      public boolean addMove (Position p) {
53              if (numberOfMoves < (size * size)) {
54                      path[numberOfMoves++] = p;
55                      board[p.getX()][p.getY()] = numberOfMoves;
56                      return true;
57              }
58              return false;
59      }
60
61      /* Pop a move from the move stack, remove it from the board */
62      public boolean removeMove () {
63              if (numberOfMoves > 0) {
64                      Position p = path[numberOfMoves - 1];
65                      /* Empty squares are marked '0' */
66                      board[p.getX()][p.getY()] = 0;
67                      path[--numberOfMoves] = null;
68                      return true;
69              }
70              return false;
71      }
72
73      public int[][] getBoard () {
74              return board;
75      }
76
77      /* Get the stack of moves comprising a knight's tour */
78      public Position[] getSolution () {
79              if (size < 5)
80                      return null;
81              addMove(initPosition);
82              if(solve(initPosition))
83                      return path;
84              return null;
85      }
86
87      /* Recursively solve a tour from a given position */
88      public boolean solve (Position p) {
89              /* If the move stack is full, the tour has been solved */
```

```
90              if (numberOfMoves == (size * size))
91                      return true;
92              /* Get every legal move and rank them using Warnsdorf's Rule */
93              Position[] possibleMoves = getPossibleMoves(p);
94              if (possibleMoves[0] == null)
95                      return false;
96              sortMoves(possibleMoves);
97              for (Position move : possibleMoves) {
98                      if (move != null) {
99                              /* Try a move */
100                             addMove(move);
101                             if (solve(move))
102                                     return true;
103                             /* Backtrack */
104                             removeMove();
105                     }
106             }
107             return false;
108     }
109
110     /* Sort a list of positions using Warnsdorf's Rule */
111     public void sortMoves (Position[] moves) {
112             int count = 0;
113             for (Position p : moves)
114                     if (p != null)
115                             count++;
116             for (int right = count; right > 0; right--)
117                     for (int i = 1; i < right; i++)
118                             if (compareMoves(moves[i-1], moves[i]) > 0)
119                                     swapMoves(i-1, i, moves);
120     }
121
122     /* Compare 2 moves using Warnsdorf's Rule */
123     public int compareMoves (Position a, Position b) {
124             /* Compare the mobilities of the knight */
125             int aCount = getPossibleMovesCount(a);
126             int bCount = getPossibleMovesCount(b);
127             if (aCount != bCount)
128                     return aCount - bCount;
129             /* Compare the mobilities of the knight on an empty board */
130             int aFree = degreesOfFreedom[a.getX()][a.getY()];
131             int bFree = degreesOfFreedom[b.getX()][b.getY()];
132             if (aFree != bFree)
133                     return aFree - bFree;
134             /* Resolve ties using a predecided element of randomness */
135             return (Math.random() < tieBreakRandomness)? 1 : -1;
```

```
136         }
137
138         /* Utility function to swap moves in the list of possible moves */
139         private static void swapMoves (int x, int y, Position[] moves) {
140                 Position t = moves[x];
141                 moves[x] = moves[y];
142                 moves[y] = t;
143         }
144
145         /* Get the list of all possible, legal moves not touching a previously
146            travelled square from a given position */
147         public Position[] getPossibleMoves (Position start) {
148                 Position[] possibleMoves = new Position[KNIGHT_MOVES.length];
149                 int i = 0;
150                 for (int[] move : KNIGHT_MOVES) {
151                         /* Generate a new */
152                         int x = start.getX() + move[0];
153                         int y = start.getY() + move[1];
154                         /* Check the legality of that move */
155                         if (isWithinBoard(x, y) && board[x][y] == 0) {
156                                 possibleMoves[i++] = new Position(x, y);
157                         }
158                 }
159                 return possibleMoves;
160         }
161
162         /* Get the number of legal moves */
163         public int getPossibleMovesCount (Position start) {
164                 int i = 0;
165                 for (Position p : getPossibleMoves(start))
166                         if (p != null)
167                                 i++;
168                 return i;
169         }
170
171         /* Check whether a position lies within the board */
172         public boolean isWithinBoard (int x, int y) {
173                 return (x >= 0 && x < size && y >= 0 && y < size);
174         }
175 }
```

```java
public class Position {
	private final int x;
	private final int y;

	/* Initialize using the coordinates on the board */
	public Position (int x, int y) {
		this.x = x;
		this.y = y;
	}

	/* Initialize using the position in algebraic notation */
	public Position (String s) {
		int x = 0;
		int i = 0;
		while (i < s.length() && Character.isAlphabetic(s.charAt(i))) {
			x = (x * 26) + Character.toLowerCase(s.charAt(i)) - 'a' + 1;
			i++;
		}
		int y = Integer.parseInt(s.substring(i));
		this.x = x - 1;
		this.y = y - 1;
	}

	public int getX () {
		return x;
	}

	public int getY () {
		return y;
	}

	public boolean equals (Position p) {
		return (p != null)
			&& (this.getX() == p.getX()) && (this.getY() == p.getY());
	}

	@Override
	public String toString () {
		return xToString(this.x) + (this.y + 1);
	}

	/* Convert a file number to its algebraic notation form */
	public static String xToString (int n) {
		int x = n + 1;
		String letters = "";
		while (x > 0) {
```

```
                letters = (char) ('a' + (--x % 26)) + letters;
                x /= 26;
            }
            return letters;
        }
}
```

```
public class KnightTour {
        public static void main (String[] args) {
            try {
                /* Parse the first command line argument as the size of the
                    board */
                int boardSize = Integer.parseInt(args[0]);
                if (boardSize <= 0)
                    throw new NumberFormatException();
                /* Parse the second command line argument as the starting
                    square
                  of the knight, written in algebraic notation */
                String initSquare = (args.length > 1)? args[1] : "a1";
                /* Parse the third command line argument as the degree of
                  randomness to be used while resolving ties */
                double randomness = (args.length > 2)?
                    Double.parseDouble(args[2])
                                                : Math.pow(0.8, boardSize) * 2;
                /* Create an object capable of solving knight's tours */
                TourSolver t = new TourSolver(boardSize, new
                    Position(initSquare), randomness);
                Position[] solution = t.getSolution();
                if (solution != null) {
                    showBoard(t.getBoard());
                    showMoves(solution);
                    if (isClosed(solution))
                        System.out.println("\nThe tour is Closed!");
                } else {
                    System.out.println("No Knight's Tours found!");
                }
            } catch (Exception e) {
                /* Handle missing or incorrectly formatted arguments */
                System.out.print("Enter an integer (> 1) as the first
                    argument, ");
                System.out.println("and a well formed chessboard coordinate as
                    the second!");
                System.out.println("                          (size,
                    startSquare * , randomness * )");
                System.out.println();
```

```
32                      System.out.println("(size     -> Solve a Tour on a (size x
                            size) board)");
33                      System.out.println("(startSquare * -> A square in algebraic
                            chess notation of the form 'fr',");
34                      System.out.println("            where f = the letter
                            representing the file(column)");
35                      System.out.println("            and  r = the number
                            representing the rank(row).)");
36                      System.out.println("(startSquare is set to 'a1' by default)");
37                      System.out.println("(randomness * -> A number between 0(no
                            randomness) and 1(even chances),");
38                      System.out.println("            determining the randomness in
                            ranking positions of");
39                      System.out.println("            the same weightage while
                            searching. A randomness of 0 will");
40                      System.out.println("            produce the same tour every
                            time, for a specific size and");
41                      System.out.println("            startSquare. Keep extremely
                            small values of randomness for");
42                      System.out.println("            very large boards.)");
43                      System.out.println("(randomness is set to 2 * (0.8)^boardSize
                            by default)");
44                      System.out.println();
45                      System.out.println("                                          <
                            * = optional arguments >");
46              }
47      }
48
49      /* Display the board, with each square marked with the move number on which
50         the knight landed on it */
51      public static void showBoard (int[][] board) {
52              String hLine = "  " + multiplyString("+-----", board.length) + "+";
53              System.out.println(hLine);
54              for (int column = board.length - 1; column >= 0; column--) {
55                      System.out.printf(" %2d ", column + 1);
56                      for (int row = 0; row < board.length; row++) {
57                              System.out.printf("| %3d ", board[row][column]);
58                      }
59                      System.out.printf("|%n%s%n", hLine);
60              }
61              System.out.print("  ");
62              for (int i = 0; i < board.length; i++) {
63                      System.out.printf(" %2s ", Position.xToString(i));
64              }
65              System.out.println();
66      }
```

```
67
68          /* Display the list of moves in the tour in algebraic notation */
69          public static void showMoves (Position[] moves) {
70                  System.out.print("\nMoves : ");
71                  String movesOut = "";
72                  for (int i = 1; i < moves.length; i++) {
73                          movesOut += (moves[i-1] + "-" + moves[i] + ", ");
74                  }
75                  System.out.println(movesOut.substring(0, movesOut.length() - 2));
76          }
77
78          /* Utility function for repeating strings */
79          public static String multiplyString (String s, int n) {
80                  String result = "";
81                  while (n --> 0)
82                          result += s;
83                  return result;
84          }
85
86          /* Check whether a tour is closed or not */
87          public static boolean isClosed (Position[] path) {
88                  int l = path.length - 1;
89                  int dX = Math.abs(path[0].getX() - path[l].getX());
90                  int dY = Math.abs(path[0].getY() - path[l].getY());
91                  return (dX == 1 && dY == 2) || (dX == 2 && dY == 1);
92          }
93  }
```

Variable Description

TourSolver		
int	size	Number of files/ranks in the chessboard
Position[]	path	Stack of moves which are part of the solved tour
int	numberOfMoves	Counter variable, number of moves made in the solved tour
int[][]	board	An integer array of integer arrays, representing a chessboard, with each square marked with the move number at which the knight lands on it
int[][]	degreesOf Freedom	An integer array of integer arrays, representing a chessboard, with each square marked with the number of possible knight moves from it (on an empty board)
Position	initPosition	The position on the board the knight starts from
double	tieBreak Randomness	The degree to which a move in the path is randomly decided
int[][]	KNIGHT_MOVES	List of legal changes in the x and y positions of a knight
TourSolver::initBoard()		
int	i, j	Counter variables
TourSolver::initDegreesOfFreedom()		
int	i, j	Counter variables
TourSolver::addMove(Position)		
Position	p	The new position to add to the path stack
TourSolver::removeMove()		
Position	p	The position popped from the path stack
TourSolver::solve()		
Position[]	possible Moves	List of possible moves that can be added to the path stack
Position	move	Current move to evaluate in the path
TourSolver::sortMoves(Position[])		
Position[]	moves	List of moves to rank using Warnsdorf's heuristic
int	count	Total number of moves in moves
int	right	Counter variable
int	i	Counter variable

TourSolver::compareMoves(Position, Position)		
Position	a, b	Positions/moves to compare using Warnsdorf's heuristic
int	aCount, bCount	Respective number of possible legal moves for a and b
int	aFree, bFree	Respective number of possible legal moves on an empty board for a and b

TourSolver::swapMoves(int, int, Position[])		
int	x, y	The indices of the moves to swap
Position[]	moves	Array of moves containing the moves to be swapped

TourSolver::getPossibleMoves(Position)		
Position	start	Position from where possible moves are to be generated
int	i	Counter variable
int[]	move	Pair of legal changes in the x and y positions of a knight
int	x, y	New x and y positions of the knight

TourSolver::getPossibleMovesCount(Position)		
Position	start	Position from where possible moves are to be generated
Position	p	Possible position

TourSolver::isWithinBoard(int, int)		
int	x, y	The x and y positions on the board to verify

Position		
int	x, y	The x and y coordinates on the board encoded by the Position

Position::this(String)		
String	s	Chess position written in algebraic notation
int	x, y	The x and y coordinates on the board
int	i	Counter variable

Position::xToString(int)		
int	n	File (x position) to convert to algebraic notation
String	letters	n expressed as a base 26 number, digits starting from (a)

`int`	`x`	Counter variable, temporarily stores the file to convert

`KnightTour::main(String[])`		
`int`	`boardSize`	Number of files/ranks in the chessboard
`String`	`initSquare`	The position on the board the knight starts from (algebraic notation)
`double`	`randomness`	The degree to which a move in the path is randomly decided
`TourSolver`	`t`	An object capable of generating *knight's tours*
`Position[]`	`solution`	The solved sequence of moves in the *knight's tour*

`KnightTour::showBoard(int[][])`		
`int[][]`	`board`	An integer array of integer arrays, representing a chessboard, with each square marked with the move number at which the knight lands on it
`String`	`hline`	Horizontal line drawn to represent board squares
`int`	`row, column, i`	Counter variables

`KnightTour::showMoves(Position[])`		
`Position[]`	`moves`	The sequence of moves to display
`int`	`i`	Counter variable

`KnightTour::multiplyString(String, int)`		
`String`	`s`	The string to multiply
`int`	`n`	The number of times to multiply `s`
`String`	`out`	The string containing `n` copies of `s`

`KnightTour::isClosed(Position[])`		
`Position[]`	`path`	The solved sequence of moves in the *knight's tour*
`int`	`l`	Index of last move in `path`
`int`	`dX, dY`	Differences in x and y coordinates of the knight between the first and last moves

This project was compiled with XeLaTeX.

All files involved in the making of this project can be found at
https://github.com/sahasatvik/Computer-Project/tree/master/XI

Satvik Saha
sahasatvik@gmail.com
https://sahasatvik.github.io