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 $(a_1 a_2 \ldots a_n)$, 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 occurences 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 $n \bmod 10$

Source Code

```java
public class Unique {
    public static void main (String[] args) {
        try {
            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) {
            System.out.println("Enter 1 argument (number[integer])!");
        }
    }

    public static boolean isUnique (long number) {
        int[] count = new int[10];
        for (long n = Math.abs(number); n > 0; n /= 10) {
            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

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,

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

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. Other-
       wise, 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.

3

Source Code

```java
public class Partition {
    public static void main (String[] args) {
        try {
            int target = Integer.parseInt(args[0]);
            if (target < 1) {
                throw new NumberFormatException();
            }
            partition(target);
        } catch (NumberFormatException | IndexOutOfBoundsException e) {
            System.out.println("Enter 1 argument (number[natural
                number])!");
        }
    }
    public static void partition (int target) {
        partition(target, target, "");
    }
    public static void partition (int target, int previousTerm, String suffix) {
        if (target == 0)
            System.out.println(suffix);
        for (int i = 1; i <= target && i <= previousTerm; i++)
            partition(target - i, i, suffix + " " + i);
    }
}
```

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.

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

The corresponding decryption function D_n is simply

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

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**

---

[4] The set of integers $K = \{0, 1, 2, \dots, n-1\}$ 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.

Source Code

```java
public class CaesarShift {
    public static void main (String[] args) {
        try {
            int shift = Integer.parseInt(args[0]) % 26;
            String plaintext = args[1].toUpperCase();
            String ciphertext = "";
            for (int i = 0; i < plaintext.length(); i++) {
                char plain = plaintext.charAt(i);
                char crypt = ' ';
                if ('A' <= plain && plain <= 'Z') {
                    crypt = numToChar(charToNum(plain) + shift);
                } else {
                    crypt = plain;
                }
                ciphertext += crypt;
            }
            System.out.println(ciphertext);
        } catch (NumberFormatException | IndexOutOfBoundsException e) {
            System.out.println("Enter 2 arguments (shift[integer],
                plaintext[text])!");
        }
    }

    public static int charToNum (char letter) {
        return Character.toUpperCase(letter) - 'A';
    }

    public static char numToChar (int number) {
        return (char) ('A' + Math.floorMod(number, 26));
    }
}
```

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 plaindrome.

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 boudary 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 maneuvering 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

```java
1   import java.util.Scanner;
2
3   public class Palindrome {
4       public static void main (String[] args) {
5           System.out.print("Enter your sentence : ");
6           String sentence = (new Scanner(System.in)).nextLine().trim();
7           boolean foundPalindrome = false;
8           System.out.println("Palindromes : ");
9           foundPalindrome |= checkWords(sentence);
10          foundPalindrome |= checkSentence(sentence);
11          if (!foundPalindrome) {
12              System.out.println("(No palindromes found!)");
13          }
14      }
15
16      public static boolean checkWords (String sentence) {
17          boolean foundPalindrome = false;
18          int start = -1;
19          int end = 0;
20          while (end < sentence.length()) {
21              while (Character.isWhitespace(sentence.charAt(++start)));
22              end = start;
23              while (end < sentence.length() &&
24                  !Character.isWhitespace(sentence.charAt(end++)));
24              String word = sentence.substring(start, end).trim();
25              if (isPalindrome(word)) {
26                  foundPalindrome = true;
27                  System.out.println(getAlphabets(word));
28              }
29              start = end - 1;
30          }
31          return foundPalindrome;
32      }
33
34      public static boolean checkSentence (String sentence) {
35          if (isPalindrome(sentence)) {
36              System.out.println("The sentence '" + sentence + "' is a
                    palindrome.");
37              return true;
38          }
39          return false;
40      }
41
42      public static boolean isPalindrome (String text) {
```

11

```
43              String rawText = getAlphabets(text).toUpperCase();
44              for (int i = 0, j = rawText.length() - 1; i < j; i++, j--) {
45                      if (rawText.charAt(i) != rawText.charAt(j)) {
46                              return false;
47                      }
48              }
49              return (rawText.length() > 1);
50          }
51
52      public static String getAlphabets (String text) {
53              String rawText = "";
54              for (int i = 0; i < text.length(); i++) {
55                      if (Character.isAlphabetic(text.charAt(i))) {
56                              rawText += text.charAt(i);
57                      }
58              }
59              return rawText;
60          }
61  }
```

Variable Description

Palindrome::main(String[])		
String	sentence	Stores the text to check for palindromes
boolean	foundPalindrome	Stores whether plaindromes have been found
Palindrome::checkWords(String)		
String	sentence	Stores the sentence to divide into words
boolean	foundPalindrome	Stores whether plaindromes 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

```java
public class SieveOfEratosthenes {
    private final int upperLimit;
    private boolean[] primes;

    public SieveOfEratosthenes (int upperLimit) {
        this.upperLimit = upperLimit;
        this.initPrimes();
    }

    public boolean[] getPrimes () {
        return primes;
    }

    public void initPrimes () {
        this.primes = new boolean[upperLimit];
        primes[0] = false;
        primes[1] = false;
        for (int i = 2; i < upperLimit; i++)
            primes[i] = true;
    }

    public void sievePrimes () {
        int prime = 2;
        while ((prime * prime) < upperLimit) {
            int multiple = prime * prime;
            while (multiple < upperLimit) {
                primes[multiple] = false;
                multiple += prime;
            }
            while (!primes[++prime]);
        }
    }
}
```

```java
public class Primes {
    public static void main (String[] args) {
        try {
            int upperLimit = Integer.parseInt(args[0]);
            if (upperLimit < 2) {
                throw new NumberFormatException();
            }
            SieveOfEratosthenes sieve = new
                SieveOfEratosthenes(upperLimit);
            sieve.sievePrimes();
```

```
10            showPrimes(sieve.getPrimes());
11        } catch (NumberFormatException | IndexOutOfBoundsException e) {
12            System.out.println("Enter 1 argument (limit[integer, >1])!");
13            System.out.println("(Primes will be dislayed up to, not
                    including 'limit')");
14        }
15    }
16
17    public static void showPrimes (boolean[] primes) {
18        int primeCount = 0;
19        int maxLength = Integer.toString(primes.length).length();
20        for (int i = 0; i < primes.length; i++) {
21            if (primes[i]) {
22                System.out.printf("%" + maxLength + "d ", i);
23                primeCount++;
24            }
25        }
26        System.out.println("\nTotal number of primes : " + primeCount);
27    }
28 }
```

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

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:

19

(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 lexographically after `names[i]`:
          A. Swap the elements at `names[i-1]` and `names[i]`.
          B. Swap the elements at `marks[i-1]` and `msrks[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

```java
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;

    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;
    }

    public boolean addMarks (String name, double score) {
        try {
            names[++lastStudent] = name;
            marks[lastStudent] = score;
            return true;
        } catch (IndexOutOfBoundsException e) {
            return false;
        }
    }

    public void displayChart () {
        System.out.println(Marksheet.multiplyString("-",
            Marksheet.SCREEN_WIDTH));
        for (int i = 0; i <= lastStudent; i++) {
            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));
            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));
```

```java
43              }

45          public void displayMaxScorers () {
46                  String maxScorers = "";
47                  double maxScore = getMaxScore();
48                  for (int i = 0; i <= lastStudent; i++) {
49                          if (marks[i] == maxScore) {
50                                  maxScorers += ", " + names[i];
51                          }
52                  }
53                  System.out.println(maxScorers.substring(1)
54                                          + " scored the highest ("
55                                          + maxScore + "/"
56                                          + maxMarks + ")");
57          }

59          public void sortByName () {
60                  for (int right = lastStudent; right > 0; right--)
61                          for (int i = 1; i <= right; i++)
62                                  if (names[i-1].compareToIgnoreCase(names[i]) > 0)
63                                          swapRecords(i, i - 1);

65          }

67          public double getMaxScore () {
68                  double max = Integer.MIN_VALUE;
69                  for (int i = 0; i <= lastStudent; i++) {
70                          max = Math.max(max, marks[i]);
71                  }
72                  return max;
73          }

75          private void swapRecords (int x, int y) {
76                  String tempName = names[x];
77                  double tempMark = marks[x];
78                  names[x] = names[y];
79                  marks[x] = marks[y];
80                  names[y] = tempName;
81                  marks[y] = tempMark;
82          }

84          public static String multiplyString (String s, int n) {
85                  String out = "";
86                  while (n --> 0)
87                          out += s;
88                  return out;
```

```
89          }
90  }
```

```java
1   import java.util.Scanner;
2   import java.util.InputMismatchException;
3
4   public class ScoreRecorder {
5       public static void main (String[] args) {
6           Scanner inp = new Scanner(System.in);
7           double maxMarks = 0.0;
8           int numberOfStudents = 0;
9           try {
10              System.out.print("Enter the maximum marks allotted for each
                    student : ");
11              maxMarks = inp.nextDouble();
12              System.out.print("Enter the total number of students : ");
13              numberOfStudents = inp.nextInt();
14              if (maxMarks <= 0) {
15                  System.out.println("Maximum marks must be positive!");
16                  System.exit(0);
17              }
18              if (numberOfStudents <= 0) {
19                  System.out.println("Number of students must be
                        positive!");
20                  System.exit(0);
21              }
22              Marksheet sheet = new Marksheet(maxMarks, numberOfStudents);
23              System.out.println("Enter " + numberOfStudents + " students'
                    names and marks : ");
24              for (int i = 0; i < numberOfStudents; i++) {
25                  String name = "";
26                  while (!inp.hasNextDouble()) {
27                      name += inp.next() + " ";
28                  }
29                  double marks = inp.nextDouble();
30                  if (marks <= 0 || marks > maxMarks) {
31                      System.out.println("Marks must be within 0.0 and
                            " + maxMarks + "!");
32                      System.exit(0);
33                  }
34                  sheet.addMarks(name.trim(), marks);
35              }
36              sheet.sortByName();
37              sheet.displayChart();
38              sheet.displayMaxScorers();
```

```
39              } catch (InputMismatchException e) {
40                  System.out.println("Invalid Input!");
41                  System.exit(0);
42              }
43          }
44  }
```

Variable Description

Marksheet		
int	SCREEN_WIDTH	Number of chracters 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

25

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.

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

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 ∈ {1, 2, ..., 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]`×`elements[2, 2]`) − (`elements[1, 2]`×`elements[2, 1]`).
3. Initialize an integer variable `determinant` to 0.
4. For each i ∈ {1, 2, ..., size}:
    (a) Call `this->getMinorMatrix(i, i)->getDeterminant()`. Store the result in `d`.
    (b) Add ($(-1)^{i+1}$ × `matrix[1, i]` × `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;

    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;
    }

    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;
    }

    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;
    public SquareMatrix (int size) {
        super(size, size);
        this.size = size;
    }

    public int getSize () {
        return this.size;
    }

    public int getDeterminant () {
```

```
13              if (this.size == 1)
14                      return getElementAt(1, 1);
15              if (this.size == 2)
16                      return (getElementAt(1, 1) * getElementAt(2, 2))
17                              - (getElementAt(1, 2) * getElementAt(2, 1));
18              int determinant = 0;
19              for (int i = 1; i <= size; i++)
20                      determinant += ((int) Math.pow(-1, 1+i)) * getElementAt(1, i)
21                                              * getMinorMatrix(1,
22                                                  i).getDeterminant();
22              return determinant;
23          }
24
25          public SquareMatrix getMinorMatrix (int row, int column) {
26              if (row < 1 || row > size || column < 1 || column > size)
27                      return null;
28              if (this.size <= 1)
29                      return new SquareMatrix(0);
30              SquareMatrix minor = new SquareMatrix(this.size - 1);
31              for (int i = 1, p = 1; p < size; i++, p++) {
32                  if (i == row)
33                          i++;
34                  for (int j = 1, q = 1; q < size; j++, q++) {
35                      if (j == column)
36                              j++;
37                      minor.setElementAt(this.getElementAt(i, j), p, q);
38                  }
39              }
40              return minor;
41          }
42  }


 1  import java.util.Scanner;
 2
 3  public class Determinant {
 4      public static void main (String[] args) {
 5          Scanner inp = new Scanner(System.in);
 6          try {
 7              System.out.print("Enter the size of the (size X size) square
                    matrix : ");
 8              int size = inp.nextInt();
 9              SquareMatrix matrix = new SquareMatrix(size);
10              System.out.println("Enter " + (size * size) + " integers : ");
11              for (int i = 1; i <= size; i++)
12                  for (int j = 1; j <= size; j++)
```

```
13                              matrix.setElementAt(inp.nextInt(), i, j);
14                      System.out.println("\nThe determinant is : " +
                            matrix.getDeterminant());
15              } catch (Exception e) {
16                      System.out.println("Invalid Input!");
17              }
18      }
19
20      public static void showMatrix (Matrix m) {
21              for (int i = 1; i <= m.getRows(); i++) {
22                      for (int j = 1; j <= m.getColumns(); j++) {
23                              System.out.printf("%4d ", m.getElementAt(i, j));
24                      }
25                      System.out.println();
26              }
27      }
28 }
```

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

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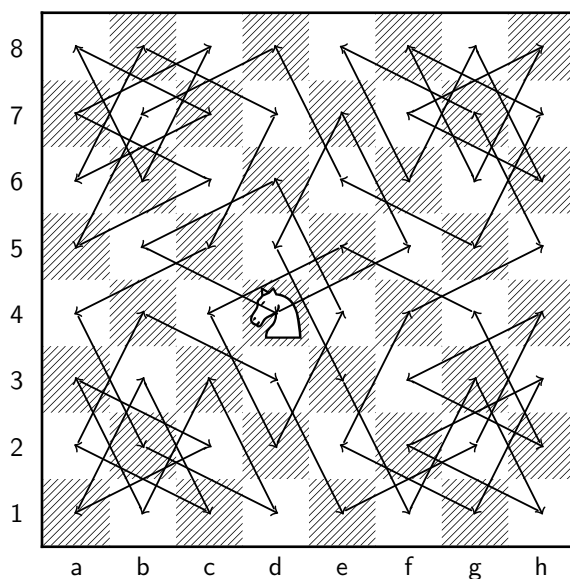
Construct a *Knight's Tour* (*open* or *closed*) on an $n \times n$ board, starting from a given square.

(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.
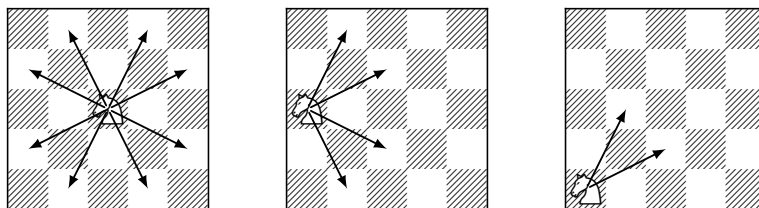

The mobility of a knight can make varies greatly with its position on the board — near the centre, it can jump to one of 8 squares while when in a corner, it can jump to only 2. On the other hand, the number of possible *sequences* of squares a knight can traverse grows extremely quickly. Although it 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*.

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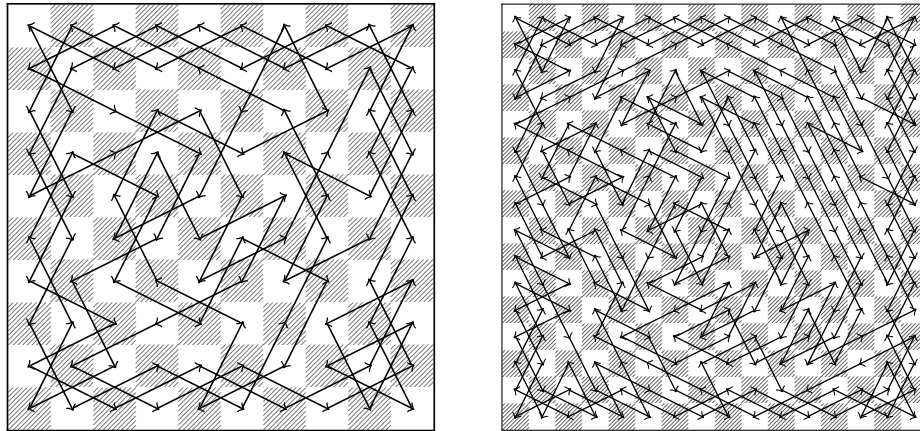
The tendency of the path to remain close to the edges of the board, where the mobility of the knight is restricted, is clearly evident.

```
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 ad 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 poition 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

```java
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;

        private static final int[][] KNIGHT_MOVES = {
                {-1, -2}, {-1, 2}, {1, -2}, {1, 2},
                {-2, -1}, {-2, 1}, {2, -1}, {2, 1}
        };

        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();
        }

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

        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;
        }

        private void initDegreesOfFreedom () {
                degreesOfFreedom = new int[size][size];
                for (int i = 0; i < size; i++)
                        for (int j = 0; j < size; j++)
                                degreesOfFreedom[i][j] = getPossibleMovesCount(new
                                        Position(i, j));
        }
```

```java
44
45        public boolean addMove (Position p) {
46                if (numberOfMoves < (size * size)) {
47                        path[numberOfMoves++] = p;
48                        board[p.getX()][p.getY()] = numberOfMoves;
49                        return true;
50                }
51                return false;
52        }
53
54        public boolean removeMove () {
55                if (numberOfMoves > 0) {
56                        Position p = path[numberOfMoves - 1];
57                        board[p.getX()][p.getY()] = 0;
58                        path[--numberOfMoves] = null;
59                        return true;
60                }
61                return false;
62        }
63
64        public int[][] getBoard () {
65                return board;
66        }
67
68        public Position[] getSolution () {
69                if (size < 5)
70                        return null;
71                addMove(initPosition);
72                if(solve(initPosition))
73                        return path;
74                return null;
75        }
76
77        public boolean solve (Position p) {
78                if (numberOfMoves == (size * size))
79                        return true;
80                Position[] possibleMoves = getPossibleMoves(p);
81                if (possibleMoves[0] == null)
82                        return false;
83                sortMoves(possibleMoves);
84                for (Position move : possibleMoves) {
85                        if (move != null) {
86                                addMove(move);
87                                if (solve(move))
88                                        return true;
89                                removeMove();
```

```java
 90                    }
 91                }
 92                return false;
 93        }

 95        public void sortMoves (Position[] moves) {
 96                int count = 0;
 97                for (Position p : moves)
 98                        if (p != null)
 99                                count++;
100                for (int right = count; right > 0; right--)
101                        for (int i = 1; i < right; i++)
102                                if (compareMoves(moves[i-1], moves[i]) > 0)
103                                        swapMoves(i-1, i, moves);
104        }

106        public int compareMoves (Position a, Position b) {
107                int aCount = getPossibleMovesCount(a);
108                int bCount = getPossibleMovesCount(b);
109                if (aCount != bCount)
110                        return aCount - bCount;
111                int aFree = degreesOfFreedom[a.getX()][a.getY()];
112                int bFree = degreesOfFreedom[b.getX()][b.getY()];
113                if (aFree != bFree)
114                        return aFree - bFree;
115                return (Math.random() < tieBreakRandomness)? 1 : -1;
116        }

118        private static void swapMoves (int x, int y, Position[] moves) {
119                Position t = moves[x];
120                moves[x] = moves[y];
121                moves[y] = t;
122        }

124        public Position[] getPossibleMoves (Position start) {
125                Position[] possibleMoves = new Position[KNIGHT_MOVES.length];
126                int i = 0;
127                for (int[] move : KNIGHT_MOVES) {
128                        int x = start.getX() + move[0];
129                        int y = start.getY() + move[1];
130                        if (isWithinBoard(x, y) && board[x][y] == 0) {
131                                possibleMoves[i++] = new Position(x, y);
132                        }
133                }
134                return possibleMoves;
135        }
```

```
136
137         public int getPossibleMovesCount (Position start) {
138                 int i = 0;
139                 for (Position p : getPossibleMoves(start))
140                         if (p != null)
141                                 i++;
142                 return i;
143         }
144
145         public boolean isWithinBoard (int x, int y) {
146                 return (x >= 0 && x < size && y >= 0 && y < size);
147         }
148 }


1  public class Position {
2          private final int x;
3          private final int y;
4
5          public Position (int x, int y) {
6                  this.x = x;
7                  this.y = y;
8          }
9
10         public Position (String s) {
11                 int x = 0;
12                 int i = 0;
13                 while (i < s.length() && Character.isAlphabetic(s.charAt(i))) {
14                         x = (x * 26) + Character.toLowerCase(s.charAt(i)) - 'a' + 1;
15                         i++;
16                 }
17                 int y = Integer.parseInt(s.substring(i));
18                 this.x = x - 1;
19                 this.y = y - 1;
20         }
21
22         public int getX () {
23                 return x;
24         }
25
26         public int getY () {
27                 return y;
28         }
29
30         public boolean equals (Position p) {
31                 return (p != null)
```

```
32                          && (this.getX() == p.getX()) && (this.getY() == p.getY());
33          }
34
35          @Override
36          public String toString () {
37                  return xToString(this.x) + (this.y + 1);
38          }
39
40          public static String xToString (int n) {
41                  int x = n + 1;
42                  String letters = "";
43                  while (x > 0) {
44                          letters = (char) ('a' + (--x % 26)) + letters;
45                          x /= 26;
46                  }
47                  return letters;
48          }
49  }
```

```
1   public class KnightTour {
2          public static void main (String[] args) {
3                  try {
4                          int boardSize = Integer.parseInt(args[0]);
5                          if (boardSize <= 0)
6                                  throw new NumberFormatException();
7                          String initSquare = (args.length > 1)? args[1] : "a1";
8                          double randomness = (args.length > 2)?
9                              Double.parseDouble(args[2])
                                                    : Math.pow(0.8, boardSize) * 2;
10                          TourSolver t = new TourSolver(boardSize, new
                                Position(initSquare), randomness);
11                          Position[] solution = t.getSolution();
12                          if (solution != null) {
13                                  showBoard(t.getBoard());
14                                  showMoves(solution);
15                                  if (isClosed(solution))
16                                          System.out.println("\nThe tour is Closed!");
17                          } else {
18                                  System.out.println("No Knight's Tours found!");
19                          }
20                  } catch (Exception e) {
21                          System.out.print("Enter an integer (> 1) as the first
                                argument, ");
22                          System.out.println("and a well formed chessboard coordinate as
                                the second!");
```

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

```
57              }
58
59      public static void showMoves (Position[] moves) {
60              System.out.print("\nMoves : ");
61              String movesOut = "";
62              for (int i = 1; i < moves.length; i++) {
63                      movesOut += (moves[i-1] + "-" + moves[i] + ", ");
64              }
65              System.out.println(movesOut.substring(0, movesOut.length() - 2));
66      }
67
68      public static String multiplyString (String s, int n) {
69              String result = "";
70              while (n --> 0)
71                      result += s;
72              return result;
73      }
74
75      public static boolean isClosed (Position[] path) {
76              int l = path.length - 1;
77              int dX = Math.abs(path[0].getX() - path[l].getX());
78              int dY = Math.abs(path[0].getY() - path[l].getY());
79              return (dX == 1 && dY == 2) || (dX == 2 && dY == 1);
80      }
81 }
```

Variable Description

TourSolver		
int	size	Number of files/ranks in the chessboard
Position[]	path	Stack of moves which are part of th 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 nuber 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	Rank (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 rank 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 X∃LATEX.

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`