# Anagram

**Problem Statement:** An anagram number is a number that can be multiplied by at least one single digit number (other than 1) to become an anagram of itself. Any anagrams formed by multiplying an anagram number by a digit are said to be generated by that anagram number. Two numbers are anagrams of each other if they can both be formed by rearranging the same combination of digits.

**Algorithm:** This program uses two methods: main() and checkAnagram()

main():

takes a number as input from the user

makes a loop run eight times changing the multiplicative values from 2 to 9

calls the checkAnagram() method passing the users input and the loop variable

prints the number if the condition satisfies

else prints "NO" if no number satisfies the condition

checkAnagram():

multiplies the two parameters passed to the function

creates array spaces to store the number of digits in each number

find the digits in the users input and stores in an array

find the digits in the multiplicated number and stores in an array

checks if the number of digits are the same in both arrays and returns

**Input/Output:**

Input:

123456789

Output:

2 4 5 7 8

Input:

100

Output:

NO

**Code:**

**import** java.util.Scanner;

**public** **class** Anagram {

**public** **static** **void** main() {

Scanner s = **new** Scanner(System.***in***);

**int** num = s.nextInt();

**boolean** flag = **false**;

**for** (**int** x = 2; x < 10; x++)

**if** (*checkAnagram*(num, x)) {

flag = **true**;

System.***out***.print(x + " ");

}

**if** (!flag)

System.***out***.println("NO");

s.close();

}

**public** **static** **boolean** checkAnagram(**int** num, **int** mul) {

**int** ana = num \* mul;

**int**[] num\_array = **new** **int**[10], ana\_array = **new** **int**[10];

**while** (num != 0) {

**int** temp = num % 10;

num /= 10;

num\_array[temp]++;

}

**while** (ana != 0) {

**int** temp = ana % 10;

ana /= 10;

ana\_array[temp]++;

}

**for** (**int** x = 0; x < 10; x++)

**if** (num\_array[x] != ana\_array[x])

**return** **false**;

**return** **true**;

}

}

# Biggest List

**Problem Statement:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | 5 | 4 | 4 | -1 | 1 | -1 | 3 | 0 | 8 |

The table above shows the content of an array A, with the indices in the top row. The array stores the pointers of a linear linked list. In this task, a pointer is simply an integer value. The pointer in the first node is stored in A[0], that is, the value A[0] indicates the location of the second node. The pointer in the second node is stored in A[A[0]], whereas the pointer in the third node is stored in A[A[A[0]]], and so on. If the pointer has value -1, it indicates end of the linked list. In the above example, value of A[0] is 2, and thus the second pointer is stored in A[2]. The value of A[2] is 4 and thus the third pointer is stored in A[4]. The value of A[4] is -1, indicating that there is no more subsequent node. The sequence of pointers is shown below and the linked list has 3 nodes.

A[0] = 2 -> A[2] = 4 -> A[4] = -1

You have received the entries of an array as described above, and you are also told that one of the entries is being replaced. However, the location of that entry, and its new value are not known. Note that it is possible that the new value is the same as the original value, that is, the array remains unchanged. As a forensic expert, you want to recover the original array. To achieve that, you want to modify a single entry, so that the modified array represents a linked list with the largest number of nodes. Consider the above example: \_ Changing the entry A[4] to 6 leads to a linked list of 4 nodes. \_ Changing the entry A[0] to 7 leads to a linked list of 4 nodes. \_ Changing the entry A[0] to 9 leads to an invalid linked list. \_ Changing the entry A[2] to 7 leads to a linked list of 5 nodes. Among all possible changes to one entry, including not changing any entry, the recovered list with 5 nodes is the largest possible. Thus, it is likely that the original entry of A[2] is 7. Here, your task is to compute the size of the largest possible recovered linked list.

**Algorithm:** This program uses three methods: main(), getInput() and getCount()

main():

uses the getInput() method to get input from the user

finds the number of jumps in the array to reach the value -1 through getCount()

finds the minimum amongst all the values

prints the maximum amount of jumps to reach a -1

getInput():

gets the users input in a single line

puts the integers in an array of ten spaces

getCount():

starts at the index of what is passed

loops until the value is not -1

the index becomes the current index’s value

returns the amount of jumps plus 1

**Input/Output:**

Input:

2 5 4 4 -1 1 -1 3 0 8

Output:

5

Input:

9 5 4 4 -1 1 -1 3 0 8

Output:

3

**Code:**

**import** java.util.Scanner;

**public** **class** BiggestList {

**static** Scanner *s* = **new** Scanner(System.***in***);

**public** **static** **void** main() {

**int** sets = *s*.nextInt();

**int**[] numbers = **null**, counts = **null**;

**for** (**int** i = 0; i < sets; i++) {

numbers = *getInput*();

counts = **new** **int**[numbers.length];

System.***out***.print("Case " + (i + 1) + ":");

**for** (**int** x = 0; x < numbers.length; x++) {

counts[x] = *getCount*(x, numbers);

}

**int** max = 0;

**for** (**int** x = 0; x < counts.length; x++) {

**if** (counts[x] > max) {

max = counts[x];

}

}

System.***out***.println(max);

System.***out***.println();

}

}

**public** **static** **int**[] getInput() {

**int**[] nums = **new** **int**[*s*.nextInt()];

*s* = **new** Scanner(System.***in***);

String line = *s*.nextLine().trim();

String[] strnums = line.split(" ");

**for** (**int** x = 0; x < strnums.length; x++) {

**try** {

nums[x] = Integer.*parseInt*(strnums[x]);

} **catch** (Exception e) {

System.***err***.println("Invalid Input");

}

}

*s*.close();

**return** nums;

}

**public** **static** **int** getCount(**int** start, **int**[] numbers) {

**int** count = 0;

**int** node = numbers[start];

**while** (node != -1) {

count++;

**if** (count > 1000) {

**return** 0;

}

**try** {

node = numbers[node];

} **catch** (Exception e) {

System.***err***.println("Invalid input");

}

}

**return** count + 1;

}

}

# Bipartite Numbers

**Problem Statement:** A bipartite number is any positive integer that contains exactly 2 distinct decimal digits s and t such that s is not 0 and all occurrences of s precede all occurrences of t. For example 44444411 is bipartite (s is 4 and t is 1), so are 41, 10000000, and 5555556. However, neither 4444114 nor 44444 are bipartite.

Notice that the large bipartite number 88888888888800000 can be nicely described as 12 8’s followed by 5 0’s. You can express any bipartite number using four numbers: m s n t. The numbers s and t are the leading and trailing digits as described above, m is the number of times the digit s appears in the bipartite number, and n is the number of times the digit t appears.

**Algorithm:** This program has 3 functions: main(), chkBipartite(long) and reduce(long)

main:

gets the input from the user

if the number is bipartite using the chkBipartite method

the number is reduced into "m s n t" form using the reduce method

the result is printed

else the number is added to the users input and checked for bipartite

chkBipartite:

an array is created for the number

's' is taken as the first number in the input

't' is taken as the last

if s is equal to t, false is returned

else

the start of the number is checked for 's'

a count variable for 's' is incremented

when the letter is not 's', the loop is broken

the end of the number is checked for 't'

a count variable for 't' is incremented

when the letter is not 't', the loop is broken

returns the sum of the counts compared to the arrays length (if the sum of the counts = arrays length, returns true, else, false)

reduce:

creates an array for 'm and 'n', which are the counts of 's' and 't' respectively

creates an array to store the number

loops through the array of the number

if the array space is equal to 's'

the value for 'm' is incremented

else the value for 'n' is incremented

returns the 'm' and 'n' array

**Input/Output:**

Input:

4444444000000000

Output:

7 4 9 0

Input:

120

Output:

1 6 2 0

**Code:**

**import** java.util.Scanner;

**public** **class** Bipartite {

**static** **int** *s* = 0, *t* = 0;

**public** **static** **void** main() {

Scanner s = **new** Scanner(System.***in***);

System.***out***.println("Enter number: ");

**long** num = s.nextInt();

**long** orignum = num;

**while** (**true**)

**if** (*chkBipartite*(num)) {

System.***out***.println(num);

**int**[] mn = *reduce*(num);

System.***out***.println(mn[0] + " " + s + " " + mn[1] + " " + *t*);

**break**;

} **else**

num += orignum;

s.close();

}

**public** **static** **boolean** chkBipartite(**long** n) {

String array[] = Long.*toString*(n).split("");

**int** scount = 0;

**int** tcount = 0;

*s* = (**int**) n;

**while** (*s* > 10)

*s* /= 10;

*t* = (**int**) n % 10;

**if** (*s* == *t*)

**return** **false**;

**for** (**int** x = 0; x < array.length; x++)

**if** (Integer.*parseInt*(array[x]) == *s*)

scount++;

**else**

**break**;

**for** (**int** x = array.length - 1; x >= 0; x--)

**if** (Integer.*parseInt*(array[x]) == *t*)

tcount++;

**else**

**break**;

**return** tcount + scount == array.length;

}

**public** **static** **int**[] reduce(**long** n) {

**int** mn[] = **new** **int**[2];

mn[0] = 0;

mn[1] = 0;

String array[] = Long.*toString*(n).split("");

**for** (**int** x = 0; x < array.length; x++)

**if** (Integer.*parseInt*(array[x]) == *s*)

mn[0]++;

**else**

mn[1]++;

**return** mn;

}

}

# Cows and Bulls

**Problem Statement:** Write a program in Java, to play the word game cows and bulls. The aim of the game is to find the opponents word (of length 3, 4 or 5 with no repeating characters). To find the word, the player guesses a word, and if a letter is in the right position, it is a bull, and if a letter is present in the word, it is a cow. Using these clues, the player must find the word. If a letter is a bull, it cannot also be a cow. The first input should be the length of the word. If it’s a 3-letter word, the number of attempt is 5. If it’s a 4-letter word, the number of guesses can go up to 10. If it’s a 5-letter word, then 15 chances.

**Algorithm:** This program has three functions: main(), checkWord(String) and getTotalTries(int)

main:

prints a random number from 3 to 5

gets the users input with the number of letters expressed in the random number generated

the word is then checked using checkWord to make sure no repeated letters exist and checked for the length

if the word passes both conditions then the user can start guessing

the total number of tries is decided by the getTotalTries method

the users input is taken

if the user gets the correct word "Congrats! The word is " with the word is displayed

two variables bullcount and cowcount are created

if the users input's letters match the original word's letters, the bullcount is incremented

if the user input's letters exist in the original word, the cowcount is incremented

the number of bulls and cows is displayed

the user gets more turns until he/she run out of them

checkWord:

the word is looped through and cross validated for any repetition of a letter

if there exists a repetition, false is returned, else true

getTotalTurns:

if the word length is 3, 5 is returned

if the word length is 4, 10 is returned

if the word length is 5, 15 is returned

**Input/Output**

Input:

Length: 4

Enter word: take

Output:

Guess 1:

from

Bulls: 0, Cows: 0

Guess 2:

boat

Bulls: 0, Cows: 2

Guess 3:

moat

Bulls: 0, Cows: 2

Guess 4:

moss

Bulls: 0, Cows: 0

Guess 5:

bake

Bulls: 3, Cows: 0

Guess 6:

bike

Bulls: 2, Cows: 0

Guess 7:

make

Bulls: 3, Cows: 0

Guess 8:

take

Congrats! The word is take.

**Code:**

**import** java.util.Scanner;

**public** **class** CowsBulls {

**public** **static** **void** main(String[] args) {

Scanner s = **new** Scanner(System.***in***);

**int** wordlength = (**int**) (Math.*random*() \* 3) + 3;

System.***out***.println("Length: " + wordlength);

System.***out***.println("Enter word: ");

String word = s.nextLine().trim();

**if** (*checkWord*(word) && word.length() == wordlength) {

**int** count = 0;

**while** (count < *getTotalTries*(wordlength)) {

System.***out***.println("Guess " + (count + 1) + ": ");

String guess = s.nextLine().trim();

**if** (guess.equals(word)) {

System.***out***.println("Congrats! The word is " + word + ".");

**break**;

} **else** {

**int** bullcount = 0, cowcount = 0;

**for** (**int** x = 0; x < word.length(); x++)

**if** (word.charAt(x) == guess.charAt(x))

bullcount++;

**for** (**int** x = 0; x < word.length(); x++)

**for** (**int** y = 0; y < guess.length(); y++)

**if** (word.charAt(x) == guess.charAt(y) && x != y)

cowcount++;

System.***out***.println("Bulls: " + bullcount + ", Cows: " + cowcount);

}

count++;

}

} **else**

System.***out***.println("Invalid Word");

s.close();

}

**public** **static** **boolean** checkWord(String word) {

**boolean** flag = **true**;

**for** (**int** x = 0; x < word.length(); x++)

**for** (**int** y = 0; y < word.length(); y++)

**if** (word.charAt(x) == word.charAt(y) && x != y)

flag = **false**;

**return** flag;

}

**public** **static** **int** getTotalTries(**int** length) {

**switch** (length) {

**case** 3:

**return** 5;

**case** 4:

**return** 10;

**case** 5:

**return** 15;

}

**return** 0;

}

}

# Cyclic Order

**Problem Statement:** Suppose that N players sit in order and take turns in a game, with the first person following the last person, to continue in cyclic order. While doing so, each player keeps track of the number of turns he or she has taken. The game consists of rounds, and in each round T turns are taken. After a round, the player who just had a turn is eliminated from the game. If the remaining players have all had the same number of turns, the game ends. Otherwise, they continue with another round of T moves, starting with the player just after the one who was most recently eliminated.

**Algorithm:** This program uses four functions: main(), getPlayers(int), countsEqual(int[], int[]) and adddeletedPlayer(int[], int)

main:

gets the number of players from the user

gets the number of turns from the user

uses getPlayers to get the players in an array

creates a new array to store the score of the players

uses a do while loop that exits when countsEqual returns false

loops from 0 till the max score

if the count mod number of players exists in deleted players

the loop is continued

the score of the player is incremented

prints “End of Round”

prints all the players leaving that which exists in the deleted list

prints all the scores leaving that which exists in the deleted list

adds the last player to the deleted list using the addDeletedPlayer method

prints “After Eliminating Player ” with the player

prints all the players leaving that which exists in the deleted list

prints all the scores leaving that which exists in the deleted list

prints “End of Game”

getPlayers:

creates a new character array with the space of the number of players

loops till the number of players from 0

assigns ‘A’ plus the loop variable to the array in the specific space

returns the array

countsEqual:

gets the elements that exist in the score array but leaves the ones that exist in the deleted list

checks if all the numbers are equal

if they are, false is returned

else, true is returned

addDeletedPlayer:

a new array is created with one more space than the deleted list

the numbers in the deleted array is added to the new array

the latest number passed is added to the new array

the new array is passed

**Input/Output:**

Input:

Enter number of players: 5

Enter turns: 17

Output:

------- End of Round -------

A B C D E

4 4 3 3 3

------- After Eliminating player B -------

A C D E

4 3 3 3

------- End of Round -------

A C D E

8 8 7 7

------- After Eliminating Player C -------

A D E

8 7 7

------- End of Round -------

A D E

13 13 13

------- After Eliminating player E -------

A D

13 13

------- End of Game --------

Input:

Enter number of players: 5

Enter turns: 3

Output:

------- End of Round -------

A B C D E

1 1 1 0 0

------- After Eliminating player C -------

A B D E

1 1 0 0

------- End of Round -------

A B D E

2 1 1 1

------- After Eliminating Player A -------

B D E

1 1 1

------- End of Game --------

**Code:**

**import** java.util.Scanner;

**public** **class** CyclicOrder {

**public** **static** **void** main(String[] args) {

Scanner s = **new** Scanner(System.***in***);

System.***out***.println("Enter number of players:");

**int** n = s.nextInt();

System.***out***.println("Enter turns:");

**int** t = s.nextInt();

**char** players[] = *getPlayers*(n);

**int** count[] = **new** **int**[n];

**int** deleted[] = **new** **int**[0];

**int** x = -1;

**int** copyt;

**do** {

copyt = t + x;

outer: **for** (**int** y = x; y < copyt; y++) {

**for** (**int** i = 0; i < deleted.length; i++)

**if** ((y + 1) % n == deleted[i]) {

copyt++;

x++;

**continue** outer;

}

count[++x % n]++;

}

System.***out***.println("\n\n------- End of Round -------\n");

outer: **for** (**int** i = 0; i < n; i++) {

**for** (**int** j = 0; j < deleted.length; j++)

**if** (i == deleted[j])

**continue** outer;

System.***out***.print(players[i] + " ");

}

System.***out***.println();

outer: **for** (**int** i = 0; i < n; i++) {

**for** (**int** j = 0; j < deleted.length; j++)

**if** (i == deleted[j])

**continue** outer;

System.***out***.print(count[i] + " ");

}

deleted = *addDeletedPlayer*(deleted, x % n);

System.***out***.println(

"\n\n-------After eliminating player " + players[deleted[deleted.length - 1]] + "-------\n");

outer: **for** (**int** i = 0; i < n; i++) {

**for** (**int** j = 0; j < deleted.length; j++)

**if** (i == deleted[j])

**continue** outer;

System.***out***.print(players[i] + " ");

}

System.***out***.println();

outer: **for** (**int** i = 0; i < n; i++) {

**for** (**int** j = 0; j < deleted.length; j++)

**if** (i == deleted[j])

**continue** outer;

System.***out***.print(count[i] + " ");

}

} **while** (*countsEqual*(count, deleted));

System.***out***.println("\n\n------- End of Game -------");

s.close();

}

**public** **static** **char**[] getPlayers(**int** n) {

**char**[] players = **new** **char**[n];

**for** (**int** x = 0; x < n; x++)

players[x] = (**char**) ('A' + x);

**return** players;

}

**public** **static** **boolean** countsEqual(**int**[] count, **int**[] deleted) {

**int** chk = 0;

**int** array[] = **new** **int**[count.length - deleted.length];

outer: **for** (**int** i = 0; i < count.length; i++) {

**for** (**int** j = 0; j < deleted.length; j++)

**if** (i == deleted[j])

**continue** outer;

array[chk++] = count[i];

}

**int** temp = array[0];

**boolean** flag = **false**;

**for** (**int** x = 0; x < array.length; x++) {

**if** (array[x] != temp) {

flag = **true**;

}

}

**return** flag;

}

**public** **static** **int**[] addDeletedPlayer(**int**[] deleted, **int** index) {

**int** newarray[] = **new** **int**[deleted.length + 1];

**int** x;

**for** (x = 0; x < deleted.length; x++)

newarray[x] = deleted[x];

newarray[x] = index;

**return** newarray;

}

}

# Date Difference

**Problem Statement:** Write a program in Java, to find the difference between 2 dates given in dd/mm/yyyy format.

Note: The order of the dates shouldn’t matter. If 20/02/2019 is given before 18/02/2019, you must still get 2 days and not -2 days. The date difference can be more than 365 days.

**Algorithm:** This program uses three functions: main(), find365(String, String) and getMonthList()

main:

gets the users input of two dates

finds the 365 values for their respective months and dates using the find365() method

if the difference in days is negative, the sign is taken out

a loop runs to add days if a leap year happens

for every year in difference 365 days are added

the result is printed

find365:

a month list is loaded using the getMonthList method

a loop adds the month's days and finally the date itself to get a total number within 365

the sum is returned

getMonthList:

an array is manually typed with the number of days within each month and returned

**Input/Output:**

Input:

21/07/2013

01/12/2017

Output:

1594 days

Input:

27/11/2019

27/11/2019

Output:

282 days

**Code:**

**import** java.util.Scanner;

**public** **class** DateDifference {

**public** **static** **void** main(String[] args) {

Scanner s = **new** Scanner(System.***in***);

System.***out***.println("Enter date: ");

String date1[] = s.nextLine().trim().split("/");

System.***out***.println("Enter another date:");

String date2[] = s.nextLine().trim().split("/");

**int** day1\_365 = *find365*(date1[0], date1[1]);

**int** day2\_365 = *find365*(date2[0], date2[1]);

**int** day\_diff = day1\_365 - day2\_365;

day\_diff = day\_diff > 0 ? day\_diff : -day\_diff;

**int** lower\_year = Integer.*parseInt*(date1[2]) < Integer.*parseInt*(date2[2]) ? Integer.*parseInt*(date1[2])

: Integer.*parseInt*(date2[2]);

**int** upper\_year = Integer.*parseInt*(date1[2]) > Integer.*parseInt*(date2[2]) ? Integer.*parseInt*(date1[2])

: Integer.*parseInt*(date2[2]);

**for** (**int** x = lower\_year; x < upper\_year; x++) {

**if** (x % 4 == 0) {

day\_diff++;

}

day\_diff += 365;

}

System.***out***.println(day\_diff + " days");

s.close();

}

**public** **static** **int** find365(String day, String month) {

**int** d = Integer.*parseInt*(day);

**int** m = Integer.*parseInt*(month);

**int** days\_in\_month[] = *getMonthList*();

**int** sum = 0;

**for** (**int** x = 0; x < m - 1; x++)

sum += days\_in\_month[x];

**return** sum + d;

}

**public** **static** **int**[] getMonthList() {

**int** months[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

**return** months;

}

}

# Determinant

**Problem Statement:** Find the determinant of any square matrix.

**Algorithm:** This program uses four functions: main(), getArray(String), getMatrixFromArray(double[]) and getDeterminant(double[][])

main:

gets the users input in a specific format

gets the information in a desired array format using getArray method

gets the matrix form using the getMatrixFromArray method

finds the determinant using the findDeterminan method

prints the determinant

getArray:

the users input is passed to this function

the values are then separated by "," and assigned to an array

the "{" and "}" characters are replaced with "" and assigned to an integer array

the array is returned

getMatrixFromArray:

the formed array is passed to this function

a two dimensional array is formed taking the square root of the original array as a length

the two dimensional array is assigned all the values of the original array

the two dimensional array is returned

getDeterminant:

the matrix is passed to this method

if the length of the matrix is 2 then the determinant is found for it and returned

else a new two dimensional array is created with one less space than the matrix

a sum variable is created to hold the determinant

the first row of the original matrix is looped through

the new matrix is assigned the values of the old matrix excluding the rows and columns of the first row element taken at the moment

the sum is added with the (-1) to the power of the loop variable multiplied with the first row element multiplied with calling getDeterminant passing the new matrix

the sum is returned

**Input/Output:**Input:

{{6,3,9},{5,7,4},{7,2,1}}

Output:

Determinant: -288

Input:

{{5,6,3,5},{3,1,4,65},{72,12,11,56},{45,5,2,98}}

Output:

Determinant: 166867

**Code:**

**import** java.util.Scanner;

**public** **class** Determinant {

**public** **static** **void** main(String[] args) {

Scanner s = **new** Scanner(System.***in***);

System.***out***.println("Enter matrix:");

String input = s.nextLine().trim();

**double** array[] = *getArray*(input);

**double** matrix[][] = *getMatrixFromArray*(array);

**double** determinant = *getDeterminant*(matrix);

System.***out***.println("Determinant: " + determinant);

s.close();

}

**public** **static** **double**[] getArray(String input) {

String elements[] = input.split(",");

**double** array[] = **new** **double**[elements.length];

**for** (**int** x = 0; x < elements.length; x++)

array[x] = Integer.*parseInt*(elements[x].replaceAll("[{,}]", ""));

**return** array;

}

**public** **static** **double**[][] getMatrixFromArray(**double**[] array) {

**int** dim = (**int**) Math.*sqrt*(array.length);

**double**[][] matrix = **new** **double**[dim][dim];

**int** count = 0;

**for** (**int** x = 0; x < dim; x++)

**for** (**int** y = 0; y < dim; y++)

matrix[x][y] = array[count++];

**return** matrix;

}

**public** **static** **double** getDeterminant(**double**[][] matrix) {

**if** (matrix.length == 2)

**return** matrix[0][0] \* matrix[1][1] - matrix[1][0] \* matrix[0][1];

**double**[][] newmat = **new** **double**[matrix.length - 1][matrix.length - 1];

**int** sum = 0, ycount = 0;

**for** (**int** x = 0; x < matrix.length; x++) {

**for** (**int** i = 1; i < newmat.length + 1; i++) {

**for** (**int** j = 0; j < matrix.length; j++)

**if** (j != x)

newmat[i - 1][ycount++] = matrix[i][j];

ycount = 0;

}

sum += Math.*pow*(-1, x) \* matrix[0][x] \* *getDeterminant*(newmat);

}

**return** sum;

}

}

# Digit Words

**Problem Statement:** A digit word is a word where, after possibly removing some letters, you are left with one of the single digits:

ZERO, ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT or NINE.

Write a program which reads in a single upper-case word (with at most fifteen letters) and determines if

it is a digit word. If the word is not a digit word you should output the word NO. If the word is a digit word you should output the digit it contains, as a number.

**Algorithm:** This program uses four functions: main(), loadDigits(), checkNumber(String[], String) and checkAllFlags(boolean[])

main:

takes in the users input and makes it into upper case

assigns the digits in English to the variable digit using the loadDigits method

the word is checked using the checkNumber method

loadDigits:

the array is manually assigned the values of the numbers

checkNumber:

the word is checked with every number and an array of flags are assigned whether the letters correspond

the array is checked whether it has all true elements using the checkAllFlags method

if all the flags are true then the number is printed as a digit

else it checks with the next number

if none of the numbers match then "NO" is printed

checkAllFlags:

takes the check variable as the first flag

if the flags are not the same false is returned

the first check variable is returned

**Input/Output:**

Input:

pounce

Output:

1

Input:

tiger

Output:

NO

**Code:**

**import** java.util.Scanner;

**public** **class** DigitWords {

**public** **static** **void** main(String[] args) {

Scanner s = **new** Scanner(System.***in***);

System.***out***.println("Enter word:");

String word = s.nextLine().trim().toUpperCase();

String digit[] = *loadDigits*();

*checkNumber*(digit, word);

s.close();

}

**public** **static** String[] loadDigits() {

String nums[] = **new** String[10];

nums[0] = "ZERO";

nums[1] = "ONE";

nums[2] = "TWO";

nums[3] = "THREE";

nums[4] = "FOUR";

nums[5] = "FIVE";

nums[6] = "SIX";

nums[7] = "SEVEN";

nums[8] = "EIGHT";

nums[9] = "NINE";

**return** nums;

}

**public** **static** **void** checkNumber(String[] digit, String word) {

**boolean** total\_flag = **false**;

**for** (**int** x = 0; x < 10; x++) {

**boolean** flag[] = **new** **boolean**[digit[x].length()];

**int** count = 0;

**for** (**int** i = 0; i < word.length(); i++)

**if** (word.charAt(i) == digit[x].charAt(count))

flag[count++] = **true**;

**if** (*checkAllFlags*(flag)) {

System.***out***.println(x);

total\_flag = **true**;

}

}

**if** (!total\_flag) {

System.***out***.println("NO");

}

}

**public** **static** **boolean** checkAllFlags(**boolean**[] flag) {

**boolean** b = flag[0];

**for** (**int** x = 0; x < flag.length; x++) {

**if** (flag[x] != b) {

**return** **false**;

}

}

**return** b;

}

}

# Egyptian Fractions

**Problem Statement:** Recall that the Egyptians represented fractions using a sum of distinct unit fractions—in other words, each fraction has numerator 1 and all the denominators are different.

For example, 3/4 = 1/2 + ¼

6/7 = 1/2 + 1/3 + 1/42

(But not 2/5 = 1/5 + 1/5, because the unit fractions must be different from each other.)

It is always possible to ﬁnd an Egyptian representation for a fraction p/q < 1.

**Algorithm:** This program uses one function: main()

main:

takes a fraction from the user

splits the numerator and denominator

finds the decimal value of the fraction

loops while the Egyptian fractions sum is not equal to the decimal value

x is assigned as 2

adds 1/x to the sum

if the sum is greater than the decimal, 1/x is subtracted

else "1/x" is printed

x is incremented

**Input/Output:**

Input:

7/9

Output:

7/9 = 1/2 + 1/4 + 1/36

Input:

3/20

Output:

3/20 = 1/7 + 1/140

**Code:**

**import** java.util.Scanner;

**public** **class** EgyptianFractions {

**public** **static** **void** main(String[] args) {

Scanner s = **new** Scanner(System.***in***);

System.***out***.println("Enter fraction: ");

String frac[] = s.nextLine().trim().split("/");

**double** numer = Double.*parseDouble*(frac[0]);

**double** denom = Double.*parseDouble*(frac[1]);

**double** decimal = numer / denom;

System.***out***.print((**int**) numer + "/" + (**int**) denom + " = ");

**double** sum = 0;

**boolean** flag = **false**;

**double** x = 2;

**while** (decimal != sum) {

sum += 1 / x;

**if** (sum > decimal) {

sum -= 1 / x;

} **else** {

**if** (flag)

System.***out***.print("+ 1/" + (**int**) x + " ");

**else** {

flag = **true**;

System.***out***.print("1/" + (**int**) x + " ");

}

}

x++;

}

s.close();

}

}

# Encryption

**Problem Statement:** A simple encryption system uses a shift process to hide a message. The value of shift can be in the range 1 to 26. For example a shift of 7 means that A=U, B=V, C=W, etc. i.e

Text: A B C D E F G H I J K L M N O P Q R S T V U W X Y Z

Code: U V W X Y Z A B C D E F G H I J K L M N O P Q R S T

First an extra space is added to the end of the string. Spaces within original text are replaced with QQ before the text is encrypted. Double Q(QQ) was selected because no English word ends or contains QQ. Additionally the coded message is printed in blocks of six characters separated by spaces. The last block might not contain six characters. The program takes the coded text(less than 100 characters), the shift value and prints the decoded original text.

**Algorithm:** This program uses two functions: main() and decode(String, int)

main:

gets the users input

get the shift value from the user

decodes the message using decode method

prints the decoded message with replacing all 'QQ' with ''

decode:

the encrypted message is passed

all of the spaces are replaced with ""

the message is looped through

if the character at a given position added with the shift is lesser than 'Z'

the character plus the shift is added to the final decrypted message

else the character plus the shift's remainder with 91 ('Z'+1) is found

the remainder plus 'A' is added to the final message

the decrypted message is returned

**Input/Output:**

Input:

UHINBY LKKQCH HYLKK

shift: 7

Output:

ANOTHER WINNER

Input:

RUIJGG EVGGBK SAGG

shift: 11

Output:

BEST OF LUCK

**Code:**

**import** java.util.Scanner;

**public** **class** Encryption {

**public** **static** **void** main(String[] args) {

Scanner s = **new** Scanner(System.***in***);

System.***out***.println("Enter input: ");

String input = s.nextLine().trim();

System.***out***.println("Enter shift: ");

**int** shift = s.nextInt();

String decoded = *decode*(input, shift);

System.***out***.println("Decoded message: " + decoded.replaceAll("QQ", " "));

s.close();

}

**public** **static** String decode(String text, **int** shift) {

String decoded = "";

text = text.replaceAll(" ", "");

**for** (**int** x = 0; x < text.length(); x++) {

**if** (text.charAt(x) + shift - 1 <= 'Z')

decoded += (**char**) (text.charAt(x) + shift - 1);

**else** {

**int** temp = ((**int**) text.charAt(x) + shift - 1) % 91;

decoded += (**char**) ('A' + temp);

}

}

**return** decoded;

}

}

# Kaprekar Number

**Problem Statement:** A Kaprekar number is a number, when squared and split in two parts, sums up to the original number. For example: 99.

992 = 9801

98 + 01 = 99

Therefore, 99 is a Kaprekar number

**Algorithm:** This program has 1 function: main

main:

takes an input from the user

the number is squared and converted to a string datatype

if the length of the squared number is divisible by two

the squared number is split into two parts - the squared number divided in two halves

if the two parts add up to the original number

prints "kaprekar number"

else prints "not a kaprekar number"

else

the squared number is split in four parts - the squared number divided in two parts with different middle points

if the corresponding squared number parts add up to the original number

prints "kaprekar number"

else prints "not a kaprekar number"

**Input/Output:**

Input:

47

Output:

Not a Kaprekar number

Input:

99

Output:

Kaprekar number!

**Code:**

**import** java.util.Scanner;

**public** **class** Kaprekar {

**public** **static** **void** main() {

Scanner s = **new** Scanner(System.***in***);

System.***out***.println("Enter number: ");

**long** num = s.nextInt();

**long** numsq = num \* num;

String numsqstr = **new** String(Long.*toString*(numsq));

**if** (numsqstr.length() % 2 == 0) {

**int** part1 = Integer.*parseInt*(numsqstr.substring(0, numsqstr.length() / 2));

**int** part2 = Integer.*parseInt*(numsqstr.substring(numsqstr.length() / 2));

**if** (part1 + part2 == num && part2 != 0) {

System.***out***.println("Kaprekar number!");

} **else** {

System.***out***.println("Not a Kaprekar number");

}

} **else** {

**int** part1 = Integer.*parseInt*(numsqstr.substring(0, numsqstr.length() / 2));

**int** part2 = Integer.*parseInt*(numsqstr.substring(numsqstr.length() / 2));

**int** part3 = Integer.*parseInt*(numsqstr.substring(0, numsqstr.length() / 2 + 1));

**int** part4 = Integer.*parseInt*(numsqstr.substring(numsqstr.length() / 2 + 1));

**if** (part1 + part2 == num && part2 != 0) {

System.***out***.println("Kaprekar number!");

} **else** **if** (part3 + part4 == num && part4 != 0) {

System.***out***.println("Kaprekar number!");

} **else** {

System.***out***.println("Not a Kaprekar number");

}

}

s.close();

}

}

# Letter Change

**Problem Statement:** A string rewriting rule is a simple instruction for changing a string. It says that every occurrence of a given character should be changed into one or more other characters. In this question we will use five rules:

* A should be changed into B
* B should be changed into AB
* C should be changed into CD
* D should be changed into DC
* E should be changed into EE

**Algorithm:** This program uses two methods: main() and replaceLetters()

main():

takes the users input and converts it into upper case

takes users input for the number of times to change the word

iteratively changes the letters by calling replaceLetters()

gets the users input to count the number of letters from the start

iterates through the words letters and increments an array which keeps a track of the letters

displays the amount of letters in the amount of spaces the users has entered

replaceLetters():

converts the word into a string array

iterates through the letters and changes the letters based on the conditions in the problem statement

combines all the array spaces into one string

returns the string

**Input/Output:**

Input:

dec

2

10

Output:

0 0 3 3 4

Input:

decade

2

15

Output:

1 1 4 5 4

**Code:**

**import** java.util.Scanner;

**public** **class** LetterChange {

**public** **static** **void** main(String[] args) {

Scanner s = **new** Scanner(System.***in***);

String word = s.nextLine().trim();

word = word.toUpperCase();

**int** n = s.nextInt();

**for** (**int** x = 0; x < n; x++)

word = *replaceLetters*(word);

**int** p = s.nextInt();

**char** letters[] = word.toCharArray();

**int** count[] = **new** **int**[5];

**for** (**int** x = 0; x < p; x++)

count[((**int**) letters[x]) - 65]++;

**for** (**int** x = 0; x < 5; x++)

System.***out***.print(count[x] + " ");

}

**public** **static** String replaceLetters(String word) {

**char** letters[] = word.toCharArray();

String lets[] = **new** String[letters.length];

**for** (**int** x = 0; x < letters.length; x++)

lets[x] = Character.*toString*(letters[x]);

**for** (**int** x = 0; x < lets.length; x++) {

**switch** (lets[x]) {

**case** "A":

lets[x] = "B";

**break**;

**case** "B":

lets[x] = "AB";

**break**;

**case** "C":

lets[x] = "CD";

**break**;

**case** "D":

lets[x] = "DC";

**break**;

**case** "E":

lets[x] = "EE";

}

}

word = "";

**for** (**int** x = 0; x < lets.length; x++) {

word += lets[x];

}

**return** word;

}

}

# Linked List

**Problem Statement:** Unlike an array, a linked list is a list that has no limit to storing. In this case the user can keep entering inputs and the linked list will automatically sort the elements.

**Algorithm:** This class uses the class Node

The Node class object has two pieces of information

The node’s value

The next Node's address

This class has 6 methods: insert(double), delete(int), get(), get(int), reverse() and split(int)

insert:

if the linked list is empty

the first node is assigned as the newly created node

the last node is assigned as the newly created node

else

if the new node's value is greater than the first node's

the new node is assigned as the first node

else if the new node's value is lesser than the last node's value

the new node is assigned as the last node

else

the node is created in between the linked list

delete:

if the index is lesser than 0

prints "no index found"

else

loops while the current node is not null

if the iteration is equal to the index

the previous node's next node is assigned as the current node's next node, therein deleting the current node

the loop is broken

else if the next node in the linked list is equal to null

prints "index not available"

the loop is broken

else the current nodes is assigned as the next node

get:

loops while the current node is not equal to null

prints the current node's data

assigns the current node as the next node in the linked list

get(with an index):

if the index is lesser than 0

prints "no index found"

else

loops while the current node is not null

if the iteration is equal to the index

prints the current node's data

the loop is broken

else if the next node in the linked list is equal to null

prints "index not available"

the loop is broken

else the current nodes is assigned as the next node

reverse:

number of nodes is found using a loop

an array data is created with number of nodes amount of spaces

data is assigned the linked lists data by iterating till count

the nth node is assigned to the (length of the array - n)th index of the data array

split:

an index is passed

the linked list after the index is stored in another linked list

the newly formed linked list is returned

**Input/Output:**

Input:

Enter input: 5

Enter input: 6

Enter input: 1

Enter input: 4

Enter input: 3

Output:

Linked List:

1.0

3.0

4.0

5.0

6.0

Input:

Enter index to delete: 2

Output:

Linked List:

1.0

3.0

5.0

6.0

Input:

Enter index for element: 1

Output:

3.0

Reversed Linked List:

6.0

5.0

4.0

1.0

**Code:**

**public** **class** LinkedList {

Node first, last, curr, prev;

**public** **void** insert(**double** val) {

Node newnode = **new** Node(val);

**if** (first == **null**) {

first = newnode;

last = newnode;

} **else** {

**if** (newnode.data < first.data) {

newnode.next = first;

first = newnode;

} **else** **if** (newnode.data > last.data) {

last.next = newnode;

last = newnode;

} **else** {

curr = first;

prev = curr;

curr = curr.next;

**while** (curr != **null**) {

**if** ((newnode.data > prev.data) && (newnode.data < curr.data)) {

prev.next = newnode;

newnode.next = curr;

**break**;

} **else** {

prev = curr;

curr = curr.next;

}

}

}

}

}

**public** **void** delete(**int** index) {

curr = first;

prev = curr;

**int** at = -1;

**if** (index < 0) {

System.***out***.println("No index found");

} **else** {

**while** (curr != **null**) {

at++;

**if** (at == index) {

prev.next = curr.next;

**break**;

}

**if** (curr.next == **null**) {

System.***out***.println("Index not available");

**break**;

} **else** {

prev = curr;

curr = curr.next;

}

}

}

}

**public** **void** get() {

// returns everything

curr = first;

**while** (curr != **null**) {

System.***out***.println(curr.data);

curr = curr.next;

}

}

**public** **void** get(**int** index) {

// returns specific data at index "index"

curr = first;

**int** at = -1;

**if** (index < 0) {

System.***out***.println("No index found");

} **else** {

**while** (curr != **null**) {

at++;

**if** (at == index) {

System.***out***.println(curr.data);

**break**;

}

**if** (curr.next == **null**) {

System.***out***.println("Index not available");

**break**;

} **else** {

curr = curr.next;

}

}

}

}

**public** **void** reverse() {

// reverses the LinkedList

curr = first;

**int** count = 0;

**while** (curr != **null**) {

count++;

curr = curr.next;

}

curr = first;

**double**[] data = **new** **double**[count];

**for** (**int** x = 0; x < count; x++) {

data[x] = curr.data;

curr = curr.next;

}

curr = first;

**for** (**int** x = count - 1; x >= 0; x--) {

curr.data = data[x];

curr = curr.next;

}

}

**public** LinkedList split(**int** index) {

curr = first;

**int** count = 0;

Node keep = **new** Node(0);

**for** (**int** x = 0; x < index; x++) {

curr = curr.next;

keep = curr;

}

**while** (curr != **null**) {

count++;

curr = curr.next;

}

curr = keep;

**double**[] data = **new** **double**[count];

**for** (**int** x = 0; x < count; x++) {

data[x] = curr.data;

curr = curr.next;

}

LinkedList l = **new** LinkedList();

**for** (**int** x = 0; x < count; x++) {

l.insert(data[x]);

}

**return** l;

}

}

# Matrix Inverse

**Problem Statement:** Find the inverse of any square matrix. An inverse is the reciprocal of the number or in our case a matrix.

Note: An inverse need not exist, for an inverse to exist, determinant must not equal 0.

The Inverse of a Matrix is written as A-1

When we multiply a matrix by its inverse we get the Identity Matrix (which is like “1” for matrices):

A × A-1  = I

The order shouldn’t matter so you must get the same thing when the inverse comes first:

A-1  × A = I

**Algorithm:** This program uses six fnuctions: main(), getMinorMatrix(double[][]), getCoFactorsMatrix(double[][]), getTranspose(double[][]), multipyDeterminant(double[][], double) and display(double[][])

main:

gets the matrix in a specific format from the user

gets the matrix in a two dimensional array form using the getArray and getMatrixFromArray methods from the Determinant class

if the determinant is not 0

gets a minor matrix using the getMinorMatrix method by passing the matrix

gets a co-factor matrix using the getCoFactorMatrix method by passing the minor matrix

gets an adjugate matrix using the getTranspose method by passing the co-factor matrix

gets an inverse matrix using the multiplyDeterminant method by passing the adjugate matrix and determinant

displays the inverse matrix using the display method

else "No inverse as determinant is 0" is displayed

getMinorMatrix:

the matrix is passed

creates a two dimensional array to store the minor matrix

creates a two dimensional array to find the determinants

loops through the two dimensional array

loops through the through the two dimensional array to assign values to the temporary matrix, skipping the current row and column values of the original matrix

finds the determinant of the temporary matrix

assigns the determinant to the new matrix of minors

returns the matrix of minors

getCoFactorMatrix:

the minor matrix is passed

loops through the two dimensional matrix

multiplies the values by (-1) raised to the power of the row number plus the column number

the matrix is returned

getTranspose:

the co-factor matrix is passed

creates a new two dimensional array to store the transpose matrix

loops through the matrix

assigns the specific row and column combination of the original matrix to the column and row combination of the transpose matrix

returns the matrix

multiplyDeterminant:

the determinant and transposed matrix is passed

prints "Inverse:"

loops through the two dimensional array and prints the matrix out

display:

takes the matrix as an input and prints it in a specific format

**Input/Output:**

Input:

{{2,1,3},{4,7,3},{2,2,5}}

Output:

Inverse:

1.12 0.04 -0.69

-0.54 0.15 0.23

-0.23 -0.08 0.38

Input:

{{1,2,3},{4,5,6},{7,8,9}}

Output:

No inverse as determinant is 0

**Code:**

**import** java.util.Scanner;

**public** **class** MatrixInverse {

**public** **static** **void** main(String[] args) {

Scanner s = **new** Scanner(System.***in***);

System.***out***.println("Enter matrix: ");

// in {{1,2,3},{4,5,6},{7,8,9}} form

String input = s.nextLine().trim();

**double**[][] matrix = Determinant.*getMatrixFromArray*(Determinant.*getArray*(input));

**double** determinant = Determinant.*getDeterminant*(matrix);

**if** (determinant != 0) {

**double**[][] minors = *getMinorMatrix*(matrix);

**double**[][] cofact = *getCoFactorsMatrix*(minors);

**double**[][] adjugate = *getTranspose*(cofact);

**double**[][] inverse = *multiplyDeterminant*(adjugate, determinant);

*display*(inverse);

} **else**

System.***out***.println("No inverse as determinant is 0");

s.close();

}

**public** **static** **double**[][] getMinorMatrix(**double**[][] matrix) {

**double**[][] newmat = **new** **double**[matrix.length][matrix.length];

**double**[][] tempmat = **new** **double**[matrix.length - 1][matrix.length - 1];

**int** xcount = 0, ycount = 0;

**for** (**int** x = 0; x < matrix.length; x++) {

**for** (**int** y = 0; y < matrix.length; y++) {

**for** (**int** i = 0; i < matrix.length; i++) {

**for** (**int** j = 0; j < matrix.length; j++)

**if** (i != x && j != y)

tempmat[xcount][ycount++] = matrix[i][j];

**if** (i != x) {

xcount++;

ycount = 0;

}

}

ycount = 0;

xcount = 0;

newmat[x][y] = Determinant.*getDeterminant*(tempmat);

}

}

**return** newmat;

}

**public** **static** **double**[][] getCoFactorsMatrix(**double**[][] matrix) {

**for** (**int** x = 0; x < matrix.length; x++) {

**for** (**int** y = 0; y < matrix.length; y++) {

matrix[x][y] = matrix[x][y] \* (**int**) Math.*pow*(-1, x + y);

}

}

**return** matrix;

}

**public** **static** **double**[][] getTranspose(**double**[][] matrix) {

**double**[][] newmat = **new** **double**[matrix.length][matrix.length];

**for** (**int** x = 0; x < matrix.length; x++) {

**for** (**int** y = 0; y < matrix.length; y++) {

newmat[x][y] = matrix[y][x];

}

}

**return** newmat;

}

**public** **static** **double**[][] multiplyDeterminant(**double**[][] matrix, **double** mult) {

**for** (**int** x = 0; x < matrix.length; x++)

**for** (**int** y = 0; y < matrix.length; y++)

matrix[x][y] = (1 / mult) \* matrix[x][y];

**return** matrix;

}

**public** **static** **void** display(**double**[][] inverse) {

System.***out***.println("Inverse:");

**for** (**int** x = 0; x < inverse.length; x++) {

**for** (**int** y = 0; y < inverse.length; y++) {

**if** (inverse[x][y] > 0)

System.***out***.print(" ");

System.***out***.printf("%.02f", inverse[x][y]);

System.***out***.print(" ");

}

System.***out***.println("\n");

}

}

}

# No E Numbers

**Problem Statement:** Everyone knows that the letter “E” is the most frequent letter in the English language. In fact, there are one hundred E’s on every page. Indeed, when spelling out integers it is interesting to see which ones do NOT use the letter “E”. For example 6030 (six thousand thirty) doesn’t. Nor does 4002064 (four million two thousand sixty four). The numbers are represented in International Place Value.

It turns out that 6030 is the 64th positive integer that does not use an “E” when spelled out and 4002064 is the 838th such number. Your task is to find the N-th such number.

**Algorithm:** This program uses two methods: main() and checkLetters();

main():

takes the users input

loops though until the users input is 0

checks if the number has no “e” when spelled in English using checkLetters()

if number has no “e”, the users input is reduced and the number is incremented

if not, the number is incremented and the loop continues

checkLetters():

makes the number into a string and splits it into its digits

loops through all of the digits

if the digits at the units, thousands, millions or billions place is 1, 3, 5, 7, 8 or 9, return true

(removes one, three, five, seven, eight, nine)

if the digits at the tens, ten thousands or ten millions place is 1, 2, 7, 8 or 9 return true

(removes teens, twenty, seventy, eighty, ninety)

if the digits at the hundreds, hundred thousands or hundred millions place is not 0, return true

(removes hundred)

returns false if nothing else is returned

**Input/Output:**

Input:

838

Output:

4002064

Input:

10

Output:

34

**Code:**

**import** java.util.Scanner;

**public** **class** NoENumbers {

**public** **static** **void** main() {

Scanner s = **new** Scanner(System.***in***);

**int** n = s.nextInt();

**int** count = 0;

**int** num = 0;

**while** (count != n) {

num++;

**if** (!*checkLetters*(num)) {

count++;

}

}

System.***out***.println(num);

}

**public** **static** **boolean** checkLetters(**int** num) {

StringBuffer sb = **new** StringBuffer(Integer.*toString*(num));

String num\_str = **new** String(sb.reverse());

**char** digits[] = num\_str.toCharArray();

**for** (**int** x = 0; x < digits.length; x++) {

**if** (x % 3 == 0)

**if** (digits[x] == '1' || digits[x] == '3' || digits[x] == '5' || digits[x] == '7' || digits[x] == '8)

|| digits[x] == '9')

**return** **true**;

**if** (x % 3 == 1)

**if** (digits[x] == '1' || digits[x] == '2' || digits[x] == '7' || digits[x] == '8' || digits[x] == '9')

**return** **true**;

**if** (x % 3 == 2 && digits[x] != '0')

**return** **true**;

}

**if** ((num / 100) % 10 != 0 || (num / 100000) % 10 != 0 || (num / 100000000) != 0) {

**return** **true**;

}

**return** **false**;

}

}

# Permutation

**Problem Statement:** A permutation of a set of numbers or alphabets (in this case), is all the possibilities of the patterns. For example: abc.

All the possible permutations are: abc, acb, bac, bca, cab, cba.

**Algorithm:** This programs has 2 functions: main() and permute()

main():

takes the letters from the user and stores it in a character array

calls permute and passes the letters

permute():

takes two inputs - x and n - the two indexes to switch in the array

if n is equal to the array's length

prints the array in a string format

returns back

switches the values of the array

calls the permute() function again passing x and (n+1)

switches the values of the array back to the original

**Input/Output:**

Input:

abcd

Output:

abcd

abdc

acbd

acdb

adcb

adbc

bacd

badc

bcad

bcda

bdca

bdac

cbad

cbda

cabd

cadb

cdab

cdba

dbca

dbac

dcba

dcab

dacb

dabc

Input:

xyz

Output:

xyz

xzy

yxz

yzx

zyx

zxy

**Code:**

**import** java.util.Scanner;

**public** **class** Permutation {

**static** **char** *lets*[];

**static** **char** *copy*[];

**public** **static** **void** main() {

Scanner s = **new** Scanner(System.***in***);

System.***out***.println("Enter letters: ");

*lets* = s.nextLine().trim().toCharArray();

*copy* = *lets*;

*permute*(0, 0);

s.close();

}

**public** **static** **void** permute(**int** x, **int** n) {

**if** (n == *lets*.length) {

System.***out***.println(String.*valueOf*(*lets*));

**return**;

}

**for** (x = n; x < *lets*.length; x++) {

**char** temp = *lets*[x];

*lets*[x] = *lets*[n];

*lets*[n] = temp;

*permute*(x, n + 1);

temp = *lets*[x];

*lets*[x] = *lets*[n];

*lets*[n] = temp;

}

}

}

# Post Fix

**Problem Statement:**

**Algorithm:** This program converts an arithmetic expression to a postfix expression

There are 4 methods: main, operators, equalsoperator, heirarchy

main()

gets the infix input from the user

makes an operator array with elements: ^, /, \*, +, -, (, )

makes a priority array with elements: 2, 1, 1, 0, 0, -1, -1, which corresponds to the operator array

makes an equation array which is the equation in a character array

creates a stack with a number of operators as spaces

number of operators is found by operators() method

loops through the whole equation with a for loop

if the character is an operator

checks using isOperator() method

if the stack is empty, the operator is pushed onto the stack

else if the operator is "(" the ( is pushed onto the stack

else the stack is looped through

if the top of the stack is a "(" and the character is ")"

the stack is popped

else if the heirarchy of the top of the stack is greater than the operator and if the top of the stack is not "("

add the operator to the postfix string

pop the stack

else add the character to the postfix

loop through the stack

if the character is "(" the stack is popped

else add the operator to the postfix string

operators()

loops through both the operator array and equation array and return the number of operations

equalsOperator():

loops through the operator array and checks if the character is an operator

heirarchy():

loops through the operator array to find the priorty of the top-of-the-stack's character

returns the priorty of the stacks top element - the current operator

**Input/Output:**

Input:

A+B/C+D\*F+G

Output:

ABC/+DF\*+G+

**Code:**

**import** java.util.Scanner;

**public** **class** PostFix {

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("Enter infix: ");

String infix = sc.nextLine().trim();

**char** oper[] = { '^', '/', '\*', '+', '-', '(', ')' };

**int** prior[] = { 2, 1, 1, 0, 0, -1, -1 };

**char** eq[] = infix.toCharArray();

Stack s = **new** Stack(*operators*(oper, eq));

String postfix = "";

outer: **for** (**int** x = 0; x < infix.length(); x++) {

**if** (*equalsOperator*(eq[x], oper)) {

**if** (s.isEmpty()) {

s.push(Character.*toString*(eq[x]));

} **else** **if** (eq[x] == '(') {

s.push(Character.*toString*(eq[x]));

} **else** {

**for** (**int** y = 0; y <= s.length; y++) {

**if** (s.peek().equals("(") && eq[x] == ')') {

s.pop();

**continue** outer;

} **else** **if** (*heirarchy*(s.peek(), prior, eq[x], oper) >= 0 && !s.peek().equals("(")) {

postfix += s.peek();

s.pop();

}

}

s.push(Character.*toString*(eq[x]));

}

} **else** {

postfix += Character.*toString*(eq[x]);

}

}

**for** (**int** x = 0; x <= s.stack.length; x++) {

**try** {

**if** (s.peek().equals("(") || s.peek().equals(")")) {

s.pop();

} **else** {

postfix += s.peek();

s.pop();

}

} **catch** (Exception e) {

**break**;

}

}

System.***out***.println(postfix);

sc.close();

}

**public** **static** **int** operators(**char** oper[], **char** eq[]) {

**int** no\_ops = 0;

**for** (**int** x = 0; x < eq.length; x++) {

**for** (**int** y = 0; y < oper.length; y++) {

**if** (eq[x] == oper[y]) {

no\_ops++;

}

}

}

**return** no\_ops;

}

**public** **static** **boolean** equalsOperator(**char** eq, **char** oper[]) {

**boolean** flag = **false**;

**for** (**int** x = 0; x < oper.length; x++) {

**if** (oper[x] == eq) {

flag = **true**;

}

}

**return** flag;

}

**public** **static** **int** heirarchy(String prev, **int** prior[], **char** now, **char** oper[]) {

**int** prevheir = 0, nowheir = 0;

**for** (**int** x = 0; x < oper.length; x++) {

**if** (Character.*toString*(oper[x]).equals(prev)) {

prevheir = prior[x];

}

}

**for** (**int** x = 0; x < oper.length; x++) {

**if** (oper[x] == now) {

nowheir = prior[x];

}

}

**return** prevheir - nowheir;

}

}

# Prime Sum

**Problem Statement:** A prime number is a whole number, greater than 1, that can only be divided by itself and the number 1. It is known that all even numbers between 4 and 3,00,00,00,00,00,00,00,000 are equal to the sum of two primes For example, 30 = 7 + 23. There are two other ways of expressing 30 as the sum of two primes, which are 11 + 19 and 13 + 17. These are the only ways of expressing 30 as the sum of two primes, since the order of the numbers in the additions does not matter.

**Algorithm:** This program has three methods: main(), calcPrimeSum() and isPrime()

main():

get the input from the user

calls the function calcPrimeSum() while passing the users input

prints the value returned by the method

calcPrimeSum():

loops a variable from 2 till half users input (to not count the sums twice)

checks if the number is prime using the isPrime() method

checks if the users input minus the number is a prime

if both are true, the number of sums are incremented

isPrime():

loops a variable from 2 to the number passed

checks if the number passed is divisible by the loop variable

if it is, the number is not prime

else, the number is a prime number

**Input/Output:**

Input:

100

Output:

6

Input:

30

Output:

3

**Code:**

**import** java.util.Scanner;

**public** **class** PrimeSum {

**public** **static** **void** main() {

Scanner s = **new** Scanner(System.***in***);

**int** num = s.nextInt();

System.***out***.println(*calcPrimeSum*(num));

s.close();

}

**public** **static** **int** calcPrimeSum(**int** num) {

**int** count = 0;

**for** (**int** x = 2; x < num / 2 + 1; x++) {

**if** (*isPrime*(x) && *isPrime*(num - x)) {

count++;

}

}

**return** count;

}

**public** **static** **boolean** isPrime(**int** num) {

**for** (**int** x = 2; x < num; x++) {

**if** (num % x == 0) {

**return** **false**;

}

}

**return** **true**;

}

}

# Shipments

**Problem Statement:** There are two coal mines, each employing a group of miners. Mining coal is hard work, so miners need food to keep at it. Every time a shipment of food arrives at their mine, the miners produce some amount of coal. There are three types of food shipments: meat shipments, fish shipments and bread shipments.

Miners like variety in their diet and they will be more productive if their food supply is kept varied. More precisely, every time a new shipment arrives to their mine, they will consider the new shipment and the previous two shipments (or fewer if there haven't been that many) and then:

* If all shipments were of the same type, they will produce one unit of coal.
* If there were two different types of food among the shipments, they will produce two units of coal.
* If there were three different types of food, they will produce three units of coal

We know in advance the types of food shipments and the order in which they will be sent. It is possible to influence the amount of coal that is produced by determining which shipment should go to which mine. Shipments cannot be divided; each shipment must be sent to one mine or the other in its entirety. The two mines don't necessarily have to receive the same number of shipments (in fact, it is permitted to send all shipments to one mine).

Your program will be given the types of food shipments, in the order in which they are to be sent. Write a program that finds the largest total amount of coal that can be produced (in both mines) by deciding which shipments should be sent to mine 1 and which shipments should be sent to mine 2.

**Algorithm:** This program uses one method: main():

main():

takes input from the user and makes it a character array

if it’s the first day one unit of coal is added

if it’s the second day

if the previous two days had the same food, one unit of coal is produced

else, two units are produced

if its any other day

if the previous three days had the dame food, one unit is produced

if there were two different foods, two units are produced

if all three days had different food, three units are produced

**Input/Output:**

Input:

MBMFFB

Output:

12

Input:

MMBMBBBBMMMMMBMB

Output:

25

**Code:**

**import** java.util.Scanner;

**public** **class** Shipments {

**public** **static** **void** main() {

Scanner s = **new** Scanner(System.***in***);

**int** lim = s.nextInt();

**for** (**int** i = 0; i < lim; i++) {

**int** unit = 0;

**int** amt = s.nextInt();

s = **new** Scanner(System.***in***);

**char**[] food = s.nextLine().trim().toCharArray();

**for** (**int** x = 0; x < amt; x++) {

**if** (x == 0 || x == 1) {

**if** (x == 0)

unit++;

**else** **if** (x == 1 && food[x] == food[x - 1])

unit++;

**else**

unit += 2;

} **else** {

**if** (food[x] == food[x - 1]) {

**if** (food[x] == food[x - 2])

unit++;

**else**

unit += 2;

} **else** {

**if** (food[x] == food[x - 2] || food[x - 1] == food[x - 2])

unit += 2;

**else**

unit += 3;

}

}

}

System.***out***.println("Mine " + (i + 1) + ": " + unit);

}

s.close();

}

}

# Shirts

**Problem Statement:** Seven shirts, with the numbers 1 to 7 embroidered on the back, are hanging on a washing line. The order of the shirts can be changed by four different operations; in each case removing one of the shirts, pushing three of the shirts along the washing line to make a gap, and replacing the removed shirt back on the washing line in the gap:

1. The leftmost shirt is temporarily removed and the adjacent three shirts pushed left.
2. The rightmost shirt is temporarily removed and the adjacent three shirts pushed right.
3. The middle shirt is temporarily removed and the leftmost three shirts pushed right.
4. The middle shirt is temporarily removed and the rightmost three shirts pushed left.

**Algorithm:** This program uses three methods: main(), reorder() and endsSame()

main():

get input from the user

calls the recursive function reorder(), passing the users input

prints the minimum number of operations to reach “1234567”

reorder():

if the order is “1234567”, then checks if the number of operations is less than the minimum

if true, assigns the number of operations as the least

if 4 of the previous operations are the same (checks using endsSame()), the recursion is broken

calls the reorder function with each of the operations

endsSame():

if the words length is less than 4, returns false

replaces a character in the word with nothing (“”) and checks the difference in length

if the difference is 4, then returns true

otherwise returns false

**Code:**

**import** java.util.Scanner;

**public** **class** Shirts {

**static** **int** *min* = 100;

**public** **static** **void** main() {

Scanner s = **new** Scanner(System.***in***);

String shirts = s.nextLine().trim();

*reorder*(shirts, "");

System.***out***.println(*min*);

s.close();

}

**public** **static** **void** reorder(String shirts, String order) {

**if** (shirts.equals("1234567")) {

**if** (order.length() < *min*) {

*min* = order.length();

}

}

**if** (*endsSame*(order)) {

**return**;

}

*reorder*(shirts.substring(1, 4) + shirts.charAt(0) + shirts.substring(4, 7), order + "A");

*reorder*(shirts.substring(0, 3) + shirts.charAt(6) + shirts.substring(3, 6), order + "B");

*reorder*(shirts.charAt(3) + shirts.substring(0, 3) + shirts.substring(4, 7), order + "C");

*reorder*(shirts.substring(0, 3) + shirts.substring(4, 7) + shirts.charAt(3), order + "D");

}

**public** **static** **boolean** endsSame(String word) {

**int** length = word.length();

**if** (length < 4) {

**return** **false**;

} **else** {

**for** (**int** x = 65; x < 69; x++) {

**int** repl\_length = word.replace(Character.*toString*((**char**) x), "").length();

**if** (length - repl\_length == 4) {

**return** **true**;

}

}

**return** **false**;

}

}

}

# Spiral

**Problem Statement:** A spiral matrix is created with a given input, the center being the number squared. For Eg: 3

1 2 3

8 9 4

7 6 5

**Algorithm:** This program has 1 function: main()

main:

gets the users input and stores its square

makes an aray to store all the numbers

the value starts at 1

two variables x and y are made to store the current index

loops while the number to be stored in the array is lesser or equal to the input squared

if the array space has no element, the value is stored

the value is incremented

y is incremented

if the y is greater than the width, the index is decremented

loops while the number to be stored in the array is lesser or equal to the input squared

if the array space has no element, the value is stored

the value is incremented

x is incremented

if the x is greater than the height, the index is decremented

loops while the number to be stored in the array is lesser or equal to the input squared

if the array space has no element, the value is stored

the value is incremented

y is decremented

if the y is lesser than 0, the index is incremented

loops while the number to be stored in the array is lesser or equal to the input squared

if the array space has no element, the value is stored

the value is incremented

x is decremented

if the x is lesser than 0, the index is incremented

the array is printed with a formatting

**Input/Output:**

Input:

8

Output:

01 02 03 04 05 06 07 08

28 29 30 31 32 33 34 09

27 48 49 50 51 52 35 10

26 47 60 61 62 53 36 11

25 46 59 64 63 54 37 12

24 45 58 57 56 55 38 13

23 44 43 42 41 40 39 14

22 21 20 19 18 17 16 15

Input:

9

Output:

01 02 03 04 05 06 07 08 09

32 33 34 35 36 37 38 39 10

31 56 57 58 59 60 61 40 11

30 55 72 73 74 75 62 41 12

29 54 71 80 81 76 63 42 13

28 53 70 79 78 77 64 43 14

27 52 69 68 67 66 65 44 15

26 51 50 49 48 47 46 45 16

25 24 23 22 21 20 19 18 17

**Code:**

**import** java.util.\*;

**public** **class** Spiral {

**public** **static** **void** main() {

Scanner s = **new** Scanner(System.***in***);

System.***out***.println("Enter number of rows:");

**int** rows = s.nextInt();

**int** num[][] = **new** **int**[rows][rows];

**for** (**int** x = 0; x < rows; x++) {

**for** (**int** y = 0; y < rows; y++) {

num[x][y] = 0;

}

}

**int** value = 1;

**int** x = 0, y = 0;

**int** rows2 = rows \* rows;

**while** (value <= rows2) {

**while** (y < rows) {

**if** (num[x][y] == 0) {

num[x][y] = value;

value++;

}

y++;

**try** {

**if** (num[x][y] != 0) {

y--;

**break**;

}

} **catch** (Exception e) {

}

}

**if** (y == rows)

y = rows - 1;

**while** (x < rows) {

**if** (num[x][y] == 0) {

num[x][y] = value;

value++;

}

x++;

**try** {

**if** (num[x][y] != 0) {

x--;

**break**;

}

} **catch** (Exception e) {

}

}

**if** (x == rows)

x = rows - 1;

**while** (y >= 0) {

**if** (num[x][y] == 0) {

num[x][y] = value;

value++;

}

y--;

**try** {

**if** (num[x][y] != 0) {

y++;

**break**;

}

} **catch** (Exception e) {

}

}

**if** (y == -1)

y++;

**while** (x >= 0) {

**if** (num[x][y] == 0) {

num[x][y] = value;

value++;

}

x--;

**try** {

**if** (num[x][y] != 0) {

x++;

**break**;

}

} **catch** (Exception e) {

}

}

**if** (x == -1)

x++;

rows--;

}

**for** (**int** i = 0; i < Math.*sqrt*(rows2); i++) {

**for** (**int** j = 0; j < Math.*sqrt*(rows2); j++) {

System.***out***.printf("%02d ", num[i][j]);

}

System.***out***.println();

}

s.close();

}

}

# Stack

**Problem Statement:** A stack is a data structure that follows the LIFO (Last In, First Out) principle.

**Algorithm:** This program 4 functions in this class: push(int), pop(), peek(), display()

push:

a number n is passed

if the stack is full, "stack overflow" is shown

else the number n is added to the stack

the top element index is incremented, the length is incremented

pop:

if the stack is empty, "stack underflow" is shown

else the top element is deleted

the top index is decremented

the length is decremented

peek:

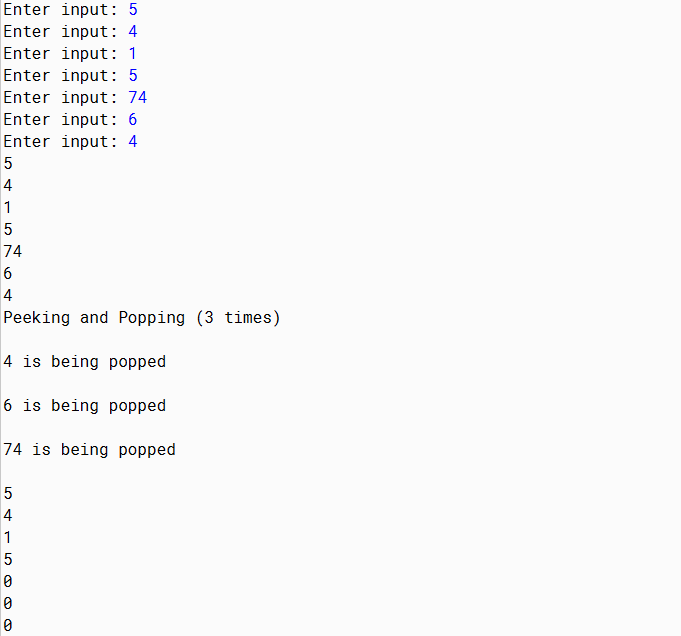
if the stack is empty, "no element" is shown

else the top element is displayed

display:

prints the full stack using a loop

**Input/Output:**



**Code:**

**public** **class** Stack {

**int** top;

**int** length;

**int** size;

**int**[] stack;

**public** Stack(**int** n) {

stack = **new** **int**[n];

top = -1;

length = 0;

size = n;

}

**public** **void** push(**int** n) {

**if** (isFull()) {

System.***err***.println("Stack Overflow");

} **else** {

stack[++top] = n;

length++;

}

}

**public** **void** pop() {

**if** (isEmpty()) {

System.***err***.println("Stack Underflow");

} **else** {

stack[top--] = 0;

length--;

}

}

**public** **boolean** isFull() {

**return** size == length;

}

**public** **boolean** isEmpty() {

**return** length == 0;

}

**public** **int** peek() {

**if** (top < 0) {

System.***err***.println("No element");

}

**return** stack[top];

}

**public** **void** display() {

**for** (**int** x = 0; x < size; x++) {

System.***out***.println(stack[x]);

}

}

}

# Stickers

**Problem Statement:** Let N be the maximum number of models Raja can label before he runs out of stickers for some digit needed to label a model. It turns out that N can be a very large number even with a small number of stickers that comes with each model. Given the set of stickers, write a program to find the number of digits in the decimal representation of N. Suppose the set of stickers has one sticker for each digit "0", "1", ..., "9". After labelling the 11th model, the remaining stickers consist of 7 stickers for the digit "1" and 10 stickers for each of the digits "0", "2", "3", ..., "9".

**Algorithm:** This program has two methods: main() and checkNumber()

main():

the users input is taken for the stickers given with each purchase

the input is stored as an array

checks using a loop if there are enough stickers to label the current model

prints the number of digits of the last model that can be numbered

checkNumber():

if the model number is not 1, more stickers are given

finds the amount of stickers required to label the current model

subtracts the number of stickers required

finds if any of the stickers have reached a negative number

if so, the model can’t be labeled

else returns that it can be labeled

**Input/Output:**

Input:

1 1 1 1 1 1 1 1 1 1

Output:

6

**Code:**

**import** java.util.Scanner;

**public** **class** Stickers {

**static** **int**[] *stickers*;

**public** **static** **void** main() {

Scanner s = **new** Scanner(System.***in***);

**int** lim = s.nextInt();

**int**[] fixed = **new** **int**[10];

s.close();

s = **new** Scanner(System.***in***);

**for** (**int** i = 0; i < lim; i++) {

String[] strnums = s.nextLine().trim().split(" ");

*stickers* = **new** **int**[strnums.length];

**for** (**int** x = 0; x < strnums.length; x++) {

*stickers*[x] = Integer.*parseInt*(strnums[x]);

}

**for** (**int** x = 0; x < *stickers*.length; x++) {

fixed[x] = *stickers*[x];

}

**int** model\_num = 1;

**while** (*checkNumber*(model\_num, fixed)) {

model\_num++;

}

System.***out***.println(Integer.*toString*(model\_num - 1).length());

}

s.close();

}

**public** **static** **boolean** checkNumber(**int** num, **int**[] fixed) {

**if** (num != 1)

**for** (**int** x = 0; x < 10; x++) {

*stickers*[x] += fixed[x];

}

**int**[] repeat = **new** **int**[10];

**boolean** flag = **true**;

**for** (**int** x = 0; x < *stickers*.length; x++) {

**int** num\_length = Integer.*toString*(num).length();

**int** replace\_num\_length = Integer.*toString*(num).replace(Integer.*toString*(x), "").length();

repeat[x] = num\_length - replace\_num\_length;

}

**for** (**int** x = 0; x < 10; x++) {

*stickers*[x] -= repeat[x];

**if** (*stickers*[x] < 0) {

flag = **false**;

}

}

**return** flag;

}

}

# Tree

**Problem Statement:**

**Algorithm:** This program creates a tree with the users input

This program has 5 methods - add, postOrder, preOrder, inOrder, search, delete and fndMaxReplace

This program uses objects from the class TreeTreeNode

The object TreeTreeNode has a value, a right TreeTreeNode and a left TreeTreeNode

The object TreeTreeNode root is the starting point for the tree

add():

the users input is passed as 'n'

if the root is not defined, the root's value is set to 'n

else if the node's value is 'n', the value is added to the right node

else if the node's value is greater than 'n', the value is added to the left node

if the left node is not null, add() is called passing the left node

else if the node's value is lesser than 'n', thw value is added to the right node

if the right node is not null, add() is called passing the right node

postOrder():

if the node's left is not null, postOrder() is called passing the left node

if the node's right is not null, postOrder() is called passing the right node

prints the node's value

preOrder():

prints the node's value

if the node's left is not null, preOrder() is called passing the left node

if the node's right is not null, preOrder() is called passing the right node

inOrder():

if the node's left is not null, inOrder() is called passing the left node

prints the node's value

if the node's right is not null, inOrder() is called passing the right node

search():

if the node is null, prints "item doesn't exist in tree"

else

if the node's value is the search, it prints "item exists in tree"

if the node's value is greater than the search element, search() is caleld passing the left node

if the node's value is lesser than the search element, search() is caleld passing the right node

delete():

a number is passed to delete

the specific node is found using recursions

a replacement node is found using the fndMaxReplace() method

if the previous node is equal to the delete node

the previous node's left is assigned to the replacement node's right

else

the previous node's right is assigned to the replacement node's left

fndMaxReplace():

a node n is passed

if n's right is not equal to null

n is defined as n's right

loops until n is equal to null

the previous node is defined as n

n is defined as n's left

n is returned

**Input/Output:**

Input:

8

5

10

3

6

9

13

1

4

12

14

Output:

Pre order: 8 5 3 1 4 6 10 9 13 12 14

Post order: 1 4 3 6 5 9 12 14 13 10 8

In order: 1 3 4 5 6 8 9 10 12 13 14

Enter number to search: 3

Item exists in tree

Enter number to delete: 8

Pre order: 9 5 3 1 4 6 10 13 12 14

Post order: 1 4 3 6 5 12 14 13 10 9

In order: 1 3 4 5 6 9 10 12 13 14

**Code:**

**public** **class** Tree {

TreeNode root = **null**;

TreeNode prev = **null**;

**public** **void** add(**int** n, TreeNode node) {

**if** (root == **null**)

root = **new** TreeNode(n);

**else** **if** (node.value == n)

add(n, node.right);

**else** **if** (node.value > n)

**if** (node.left == **null**)

node.left = **new** TreeNode(n);

**else**

add(n, node.left);

**else** **if** (node.value < n)

**if** (node.right == **null**)

node.right = **new** TreeNode(n);

**else**

add(n, node.right);

}

**public** **void** postOrder(TreeNode n) {

**if** (n.left != **null**)

postOrder(n.left);

**if** (n.right != **null**)

postOrder(n.right);

System.***out***.print(n.value + " ");

}

**public** **void** preOrder(TreeNode n) {

System.***out***.print(n.value + " ");

**if** (n.left != **null**)

preOrder(n.left);

**if** (n.right != **null**)

preOrder(n.right);

}

**public** **void** inOrder(TreeNode n) {

**if** (n.left != **null**)

inOrder(n.left);

System.***out***.print(n.value + " ");

**if** (n.right != **null**)

inOrder(n.right);

}

**public** **void** search(TreeNode n, **int** num) {

**if** (n != **null**)

**if** (n.value == num) {

System.***out***.println("Item exists in tree");

**return**;

} **else** **if** (n.value > num)

search(n.left, num);

**else**

search(n.right, num);

**else**

System.***out***.println("Item doesn't exist in tree");

}

**public** **void** delete(TreeNode n, **int** num) {

**if** (n != **null**) {

**if** (n.value == num) {

TreeNode temp = fndMaxReplace(n);

n.value = temp.value;

**if** (prev != n) {

**if** (temp.right != **null**)

prev.left = temp.right;

**else**

prev.left = **null**;

} **else** **if** (temp.left != **null**)

prev.right = temp.left;

**else**

prev.right = **null**;

} **else** **if** (n.value > num)

delete(n.left, num);

**else**

delete(n.right, num);

} **else**

System.***out***.println("Number not found");

}

**public** TreeNode fndMaxReplace(TreeNode n) {

prev = n;

**if** (n.right != **null**)

n = n.right;

**while** (n.left != **null**) {

prev = n;

n = n.left;

}

**return** n;

}

}