Find maximum and minimum

Problem

Given an array of integers, find the maximum and minimum of the array.

Constraint

Find the answer in minimum number of comparisons.

Brute force

We can keep two variables named max and min. We can iterate over the list and compare each number with the min and the max, if the number is greater than the max update max, if the number is less than the min, update the min. In this brute force solution the number of comparison is 2*n.

Better solution

If rather than comparing each number with max and min, we can first compare the numbers in pair with each other. Then compare the larger number with max and compare the smaller number with min. in this way the number of comparison for a pair of numbers are 3. So number of comparisons are 1.5 *n.

Code

```
/*
For problem and solution description please visit the link below
http://www.dsalgo.com/2013/02/FindMinMax.php.html
*/

package com.dsalgo;

public class FindMinMax
{

 public static void main(String[] args)
 {
  int[] arr = {4, 3, 5, 1, 2, 6, 9, 2, 10, 11};
  int max = arr[0];
  int min = arr[0];
  int i = 0;
  for (; i < arr.length / 2; i++)
  {
   int num1 = arr[i * 2];
   int num2 = arr[i * 2 + 1];
   if (num1 >= num2)
   {
    if (num1 > max)
     max = num1;
    if (num2 < min)
     min = num2;
   }
   else
   {
    if (num2 > max)
     max = num2;
    if (num1 < min)
     min = num1;
   }
  }
  if (i * 2 < arr.length)
  {
   int num = arr[i * 2];
   if (num > max)
    max = num;
   if (num < min)
    min = num;
  }
  System.out.println("maximum= " + max);
  System.out.println("minimum= " + min);
 }

}
```

Further discussion

Now let's think what will happen if instead of two numbers we take three numbers and then first find the largest and smallest of them and then compare these two numbers with max and min. To find the largest and smallest among 3 numbers, we need 3 comparisons in worst case and 2 comparisons in best case. The average case is 2.5. So for 3 numbers we need total 5 comparisons in worst case and 4.5 in average case. So in worst case comparisons per number is 1.6 and average case 1.5. Similarly we can derive that the worst case comparison is never less than 1.5. and best and average case is equal in case of taking numbers in pair.

Divide and conquer method

In this approach we are dividing the list in two parts, each part is recursively providing the min and max of that part and then two min max are compared to decide the ultimate min and max. Recursively when the array is reduced to only a single element then the element itself become min and max.

Code

```
/*
For problem and solution description please visit the link below
http://www.dsalgo.com/2013/02/FindMinMax.php.html
*/
```

```
package com.dsalgo;

public class FindMinMax2
{
 public static void main(String[] args)
 {
  int[] arr = {4, 3, 5, 1, 2, 6, 9, 2, 10, 11, 12};
  MinMax result = findMinMaxRecursive(arr, 0, arr.length - 1);
  System.out.println("maximum= " + result.max);
  System.out.println("minimum= " + result.min);
 }

 private static MinMax findMinMaxRecursive(int[] arr, int i, int j)
 {
  if (i > j)
   return null;
  if (i == j)
   return new MinMax(arr[i], arr[i]);
  else
  {
   MinMax left;
   MinMax right;
   left = findMinMaxRecursive(arr, i, (i + j) / 2);
   right = findMinMaxRecursive(arr, (i + j) / 2 + 1, j);
   if (left == null)
    return right;
   else if (right == null)
    return left;
   else
   {
    return new MinMax(Math.min(left.min, right.min), Math.max(
      left.max, right.max));
   }
  }
 }
}

class MinMax
{
 public int min;
 public int max;

 public MinMax(int min, int max)
 {
  this.min = min;
  this.max = max;
 }
}
```

Make larger number

Problem

Given a number whose digits are unique, find the next larger number that can be formed with those digits.

Solution

If all the digits are in decreasing order from left to right, then no larger number is possible as that is the largest number possible with those digits. If they are not in that order then a larger number is possible. We need to find the right most digit which is larger than its immediate left. Then we can bring the larger digit in place of the smaller digit and then arrange the remaining digits in increasing order. For example, if the number is 3784, 8 is the right most digit which is larger than it's left (7). We first replace 7 with 8, so it becomes 38.., then arrange the 7 and 4 in increasing order. so it becomes 3847

Code

```
/*
For problem and solution description please visit the link below
http://www.dsalgo.com/2013/02/NextLargerNumber.php.html
 */

package com.dsalgo;

import java.util.ArrayList;
import java.util.List;

public class NextLargerNumber {

 /**
  * given a number whose digits are unique find the next bigger number formed
  * by those digits
  *
  * @param args
  */
 public static void main(String[] args) throws Exception {
```

```
  System.out.println("5963=>" + getNextLarger(5963));
  System.out.println("3784=>" + getNextLarger(3784));
  System.out.println("9531=>" + getNextLarger(9531));
  System.out.println("1234=>" + getNextLarger(1234));
  System.out.println("3=>" + getNextLarger(3));
 }

 private static int getNextLarger(int decimalNumber) {
  List digits = numberToDigits(decimalNumber);
  int rightBiggerIndex = -1;
  for (int i = digits.size() - 1; i > 0; i--) {
   if (digits.get(i) > digits.get(i - 1)) {
    rightBiggerIndex = i;
    break;
   }
  }
  if (rightBiggerIndex != -1) {
   swap(digits, rightBiggerIndex, rightBiggerIndex - 1);
   sort(digits, rightBiggerIndex, digits.size());
  }
  return digitsToNumber(digits);

 }

 private static List numberToDigits(int number) {
  List digits = new ArrayList<>();
  while (number > 0) {
   digits.add(0, number % 10);
   number /= 10;
  }
  return digits;
 }

 private static int digitsToNumber(List digits) {
  int number = 0;
  for (Integer digit : digits) {
   number *= 10;
   number += digit;
  }
  return number;
 }

 private static void sort(List digits, int startIndex, int endIndex) {
  if (startIndex == endIndex)
   return;
  for (int k = startIndex; k < endIndex - 1; ++k)
   for (int l = startIndex + 1; l < endIndex; ++l) {
    if (digits.get(k) > digits.get(l))
     swap(digits, k, l);
   }

 }

 private static void swap(List digits, int i, int j) {
  Integer temp = digits.get(i);
  digits.set(i, digits.get(j));
  digits.set(j, temp);
 }
}
```

Next larger palindrome

Problem

Given an integer find the immediate larger integer that that which is a palindrome, example 1234212 -> 1234321, 345676 -> 346643.

Solution

Let the integers' digit be abcdef. As number of digits are even we will divide it in two parts, abc and def. Now we reverse first part and it becomes cba. if cba is greater than def then abccba is the answer. If it is smaller we increment the first part and it becomes (abc+1)=suppose xyz, so the answer would be xyzzyx.
Now let's check what happens when number of digits are odd. Let the integer be abcdefg. We divide it into 3 parts. abc, d, efg. if cba is greater than efg then the answer is abcdcba. If it is smaller then abcd is incremented by 1. Suppose (abcd+1)=wxyz. Then the answer is wxyzyxw.

Code

```
public class NextPalindrome
{
 public static void main(String[] args)
 {
  System.out.println(nextPalindrome(112100));
 }
```

```
 public static int nextPalindrome(int num)
 {
  return nextPalindrome(num,true);
 }
 private static int nextPalindrome(
    int num,boolean firstTime)
 {
  String numString=""+num;
  int leftEndIndex=-1;
  int rightStartIndex=-1;
  boolean isOdd=numString.length()%2==1;
  if(isOdd)
  {
   leftEndIndex=numString.length()/2;
   rightStartIndex=leftEndIndex+1;
  }
  else
  {
   leftEndIndex=rightStartIndex=numString.length()/2;

  }
  String leftHalf=numString.substring(0,leftEndIndex);
  String rightHalf=numString.substring(rightStartIndex);

  String leftReversed=new StringBuffer(leftHalf).
    reverse().toString();
  String palindrome=null;
  if(Integer.parseInt(leftReversed)>Integer.parseInt
    (rightHalf)||!firstTime)
  {
   if(isOdd)
    palindrome=leftHalf+numString.charAt(leftEndIndex)+
    leftReversed;
   else
    palindrome=leftHalf+leftReversed;
   return Integer.parseInt(palindrome);
  }
  else
  {
   if(isOdd)
   {
    String leftAndMiddle=leftHalf+numString.charAt(
    leftEndIndex);
    int incrementedLeft=Integer.parseInt(leftAndMiddle)+1;
    return nextPalindrome(Integer.parseInt(incrementedLeft+
    rightHalf),false);
   }
   else
   {
    int incrementedLeft=Integer.parseInt(leftHalf)+1;
    return nextPalindrome(Integer.parseInt(incrementedLeft+
    rightHalf),false);
   }
  }

 }
}
```

Least difference in array

Problem

Find the least difference between any two elements of an integer array.

Solution

First we can sort the elements of the array. Then we can iterate over the array and find difference between consecutive elements. The least of them would be the least differences between any two elements of the array. The complexity will be O(nlogn) because of the sorting step.

Code

```
import java.util.Arrays;


public class LeastDifference
{
 public static void main(String[] args)
 {
  int[]arr={64,57,2,78,43,73,53,86};
  Arrays.sort(arr);
```

```
  int minDiff=Integer.MAX_VALUE;
  for(int i=0;i < arr.length-1;++i)
  {
   int diff=Math.abs(arr[i]-arr[i+1]);
   if(diff < minDiff)
    minDiff=diff;
  }
  System.out.println(minDiff);
 }
}
```

Print matrix spiral

Problem

Print a matrix in spiral fashion.

Solution

We will first print the periphery of the matrix by the help of 4 for loops. Then recursively call this function to do the same thing with inner concentric rectangles. We will pass this information by a variable named depth, which will tell how many layers from outside should be ignored.

Code

```
public class PrintMatrixSpiral
{
 public static void main(String[] args)
 {
  int[][] matrix =
  {
  { 3, 4, 5, 6, 2, 5 },
  { 2, 4, 6, 2, 5, 7 },
  { 2, 5, 7, 8, 9, 3 },
  { 2, 4, 7, 3, 5, 8 },
  { 6, 4, 7, 3, 5, 7 } };

  printSpiral(matrix);
 }

 public static void printSpiral(int[][] matrix)
 {
  printSpiral(matrix, 0);
 }

 private static void printSpiral(int[][] matrix, int depth)
 {
  if (matrix == null && matrix.length == 0)
   return;
  int rows = matrix.length;
  int cols = matrix[0].length;
  if (2 * depth > Math.min(rows, cols))
   return;
  for (int i = depth; i < cols - depth - 1; ++i)
  {
   System.out.print(matrix[depth][i] + ",");
  }
  for (int i = depth; i < rows - depth - 1; ++i)
  {
   System.out.print(matrix[i][cols - depth - 1] + ",");
  }
  for (int i = rows - depth; i > depth; --i)
  {
   System.out.print(matrix[rows - depth - 1][i] + ",");
  }
  for (int i = rows - depth - 1; i > depth; --i)
  {
   System.out.print(matrix[i][depth] + ",");
  }
  printSpiral(matrix, ++depth);
 }
}
```

Move zeros to the right with minimum swap

Problem

You are given an integer array which contains some zeros. Move the zeros to the right side of the array with minimum number of swaps. The order of the original array can be destroyed.

Solution

We can do this in at most n/2 swaps. We move one index from left to right side and another from right to left side. If we get a non zero number at right index and zero in the left index then we swap the numbers. If in the right side we get a zero we just move the pointer to the left side and if we get a non zero in the left index then we move it to the right side. We continue this process till the two pointers meet.

Code

```
public class MoveZeroesToRight
{
 public static void main(String[] args)
 {
  int[] arr =
  { 0, 5, 0, 2, 3, 0, 3, 5, 0 };
  moveZeroesToRight(arr);
  for (int num : arr)
   System.out.print(num + ", ");
  System.out.println();
 }

 private static void moveZeroesToRight(int[] arr)
 {
  int start = 0;
  int end = arr.length - 1;
  while (start < end)
  {
   if (arr[start] == 0 && arr[end] != 0)
   {
    int temp = arr[start];
    arr[start] = arr[end];
    arr[end] = temp;
    start++;
    end--;
   } else
   {
    if (arr[start] != 0)
     start++;
    if (arr[end] == 0)
     end--;
   }
  }
 }

}
```

Find repetition multiple sorted arrays

Problem

Find repetition in multiple sorted arrays without using extra space. There are k sorted arrays. We have to find whether some numbers are repeating in any one of those arrays. Here k is constant and we are allowed extra space in the order of k as they are constant. The constraint is not to use extra space in order of the length of the arrays.

Solution

We will implement a solution which has complexity of O(n) and using O(k) extra space. At first we scan through individual arrays to find for repetitions. As the arrays are sorted, if there are repetitions, the numbers will be side by side. So we always keep on checking two adjacent elements and report the repetition.
Now the complex case is to find out repetition across the arrays. As the arrays are sorted we try to do a merge of the arrays like a merge sort. But only exception is we don't store the resultant array. We just keep checking the front of each arrays. If any repetition is found in the front of the k arrays we report them and move the corresponding arrays' pointer to the next element. If no repetition is found in the front row we move the pointer of the minimum number just like a merge sort. When each arrays last element is reached we conclude the search. In this way as we always are bothered about the front of each array, our extra space does not grow in the order of total elements. It is dependent on the number of front elements only, i.e, order of k.

Code

```
public class FindRepetitionWithoutExtraSpace
{

 /**
  * @param args
  */
 public static void main(String[] args)
 {
  int[][]arr={{8,12,13,16,17,22,24,29},
    {4,8,14,16,18,23},
    {33,36,37,44,95,126},
    {5,7,15,18}};
  findRepetition(arr);
 }

 public static void findRepetition(int[][] arr)
 {
  int index[]=new int[arr.length];
  int frontNumber[]=new int[arr.length];
  int length[]=new int[arr.length];
  for(int i=0;i<arr.length;++i)
  {
   length[i]=arr[i].length;
  }
  boolean modified=true;
  while(modified)
  {
   modified=false;
   for(int i=0;i<arr.length;++i)
   {
    if(index[i]<length[i])
    {
     modified=true;
     frontNumber[i]=arr[i][index[i]];
    }
    else
    {
     frontNumber[i]=Integer.MAX_VALUE;
    }
   }
   int min=frontNumber[0];
   int minIndex=0;
   for(int i=1;i<arr.length;++i)
   {
    if(frontNumber[i]==min)
    {
     if(frontNumber[i]!=Integer.MAX_VALUE)
      System.out.println(frontNumber[i]);
     index[i]++;
    }
    else if(frontNumber[i]<min)
    {
     min=frontNumber[i];
     minIndex=i;
    }
   }
   index[minIndex]++;
  }
 }
}
```

Largest sum subarray

Problem

Given an integer array find the subarray which has the largest sum.

Solution

We will keep two pointers at the start of the array. We will keep on incrementing the end pointer and calculate the current sum. If at any index current sum is larger than the maximum sum we will update the maximum sum and the corresponding indexes. If the current sum is zero or negative at any point, that means current subarray will not be a part of the maximum sum subset. In this scenario we move both the pointer to the next index and continue the process. The complexity of this algorithm is O(n) because the while loop can be looped at most 2*n times

Code

```
public class LargestSumSubArray
{
 public static void main(String[] args)
 {
  int[] arr =
  { 4, 3, -5, 0, 6, -8, 12, 3, -9, 2, 5, 8, -3, 4, 8, 0, 3, -3, -5, -9,
    4, 2 };
  maxSumSubArray(arr);
 }

 private static void maxSumSubArray(int[] arr)
 {
  int currentStart = 0;
  int currentEnd = 0;
  int currentSum = 0;
  int maxStart = 0;
  int maxEnd = 0;
  int maxSum = 0;
  while (currentEnd != arr.length)
  {
   currentSum += arr[currentEnd];
   if (currentSum > maxSum)
   {
    maxSum = currentSum;
    maxStart = currentStart;
    maxEnd = currentEnd;
   }
   if (currentSum <= 0)
   {
    currentStart = currentEnd + 1;
    currentSum=0;
   }
   currentEnd++;
  }
  System.out.println("Maximum sum = " + maxSum);
  System.out.println("Indexes (" + maxStart + "," + maxEnd + ")");
 }
}
```

Search in sorted matrix

Problem

Given a matrix whose rows and columns are sorted, search for an element in that matrix.

Solution

We start from the right top element of the array. If the search element is greater than the element we move below, if it is less we move to the left. And we continue the above process In this way either we will find the element or will reach to a position where no left or bottom is possible. In an N*M matrix the search complexity is O(N+M). In N*N matrix the complexity is O(2*N) = O(N)

Code

```
public class SearchInSortedMatrix
{
 public static void main(String[] args)
 {
  int[][] matrix =
  {
  { 5, 7, 8, 9 },
  { 6, 9, 11, 13 },
  { 7, 11, 12, 14 },
  { 8, 13, 16, 17 } };
```

```
  boolean result = contains(matrix, 14);
  System.out.println(result);

 }

 private static boolean contains(int[][] matrix, int k)
 {
  int row = matrix.length;
  int col = matrix[0].length;
  int currentRow = 0;
  int currentCol = col - 1;
  while (currentRow != row && currentCol != -1)
  {
   if (matrix[currentRow][currentCol] == k)
    return true;
   else if (matrix[currentRow][currentCol] > k)
    currentCol--;
   else
    currentRow++;
  }
  return false;
 }
}
```

Find Kth largest element in sorted matrix

Problem

Given a 2d array or matrix which is sorted by its rows and columns. Find the kth largest element from this matrix.

Solution

We will first try to find out the solution of the problem of rearranging the matrix. If one element is removed from the matrix, how can we rearrange the matrix so that the original properties of the matrix remain same, that is, sorted by its rows and columns. As the matrix is sorted by rows and columns, take any element, its top and left element will be less than the element and right and bottom element will be greater than the element. lets assume the element is a. If a is removed from the matrix, we will check left and top. Let us assume left is greater than top. If we put left in place of a,
left<a and a<right => left<right, which maintains the row wise sorting.
as a > left, bottom > a => bottom > left
and we already assumed that left>top
So these two equation proves that column wise sorting is also maintained. Similar equations will arise if top is greater than left and we replace the vacant place with top. So we can replace the removed element with the greater of top and left. and then continue the process to the next removed place till there is nothing more to remove. So this rearrange occurs in an N*M matrix in O(N+M) complexity.

Now to solve the original problem we just remove the right, bottom element k times and rearrange after every removal. As the right, bottom element is highest in the matrix, after removing k-1 times we can find the kth largest element in that position.
The total complexity will become O(K(M+N))
In an N*N matrix it is O(K(2N)) = O(KN)

Code

```
public class KthLargestSortedMatrix
{
 public static void main(String[] args)
 {
  int[][] matrix =
  {
  { 5, 7, 8, 9 },
  { 6, 9, 10, 13 },
  { 7, 11, 12, 15 },
  { 8, 13, 16, 17 } };
  int result = findKthLargest(matrix, 8);
  System.out.println(result);
 }

 private static int findKthLargest(int[][] matrix, int k)
```

```
 {
  for (int i = 0; i < k - 1; ++i)
   reArrange(matrix, matrix.length - 1, matrix[0].length - 1);
  return matrix[matrix.length - 1][matrix[0].length - 1];
 }

 private static void reArrange(int[][] matrix, int row, int col)
 {
  int newRow = 0;
  int newCol = 0;
  if (row == 0 && col == 0)
  {
   matrix[row][col] = Integer.MIN_VALUE;
   return;
  } else if (row == 0)
  {
   newRow = row;
   newCol = col - 1;
  } else if (col == 0)
  {
   newRow = row - 1;
   newCol = col;
  } else if (matrix[row][col - 1] > matrix[row - 1][col])
  {
   newRow = row;
   newCol = col - 1;
  } else
  {
   newRow = row - 1;
   newCol = col;
  }
  matrix[row][col] = matrix[newRow][newCol];
  reArrange(matrix, newRow, newCol);
 }

}
```

Find largest palindrome iterative

Problem

A string is called palindrome when it is same while reading character by character from left side or right side. Given any arbitrary string find out the largest substring in it which is a palindrome.

Brute force

A string of length n has O(n^2) substring. This is because startIndex can vary from 0 to n and endIndex can vary from startIndex to n. Now to check whether each of these substrings is a palindrome or not we need to have O(n) operation. So overall complexity would be O(n^3).

Better Solution

We will use iterative approach to solve this problem in O(n^2) time complexity using constant amount of extra space. What we will improve upon the brute force is the time to check whether the chosen substring is a palindrome or not. Instead of a O(n) time to check for palindrome we will check it in O(1) time. This will be done by using information gathered during choosing the substring. We will always chose a substring for palindrome check which has an inner palindrome just under the peripheral. So when we check the characters of two ends we can conclusively say whether the complete substring is a palindrome or not. For doing so we will chose a substring of length one which is axiomatically palindrome. Then we will grow out radially. At any point if we find a non palindrome we will stop that center and move to the next center. For even number of letters we need to chose center as the gap between two letters. For this we have chosen total 2n-1 centers to check for. While finding palindrome whenever a radius of palindrome is greater than previously stored maximum we will update the center and radius for largest palindrome.

Code

```
public class LargestPalindromeIterative
{
```

```
 public static void main(String[] args)
 {
  String str = "abccbabacbcacba";
  String result = findLargestPalindrome(str);
  System.out.println(result);
 }

 private static String findLargestPalindrome(String str)
 {
  if (str == null || str.length() == 0)
   return "";
  int centers = 2 * str.length() - 1;
  int radii = str.length() - 1;
  int maxCenter = 0;
  int maxRadius = 0;
  for (int center = 0; center < centers; ++center)
  {
   for (int radius = 0; radius <= radii; ++radius)
   {
    if (center - radius < 0 || center + radius >= centers)
    {
     break;
    } else if ((center + radius) % 2 == 1)
    {
     continue;
    } else if (str.charAt((center - radius) / 2) != str
      .charAt((center + radius) / 2))
    {
     break;
    } else
    {
     if (radius > maxRadius)
     {
      maxRadius = radius;
      maxCenter = center;
     }
    }

   }
  }

  return str.substring((maxCenter - maxRadius) / 2,
    (maxCenter + maxRadius) / 2 + 1);

 }
}
```

Reverse the words of a sentence

Problem

Reverse the words in a given sentence. Words are always delimited by spaces. For example if the given word is "reverse words of a sentence". The output will be "sentence a of words reverse"

Solution

Reverse the complete sentence and then reverse every part of sentence which is delimited by spaces.

Code

```
public class ReverseWordsInSentence
{
```

```
 public static void main(String[] args)
 {
  String str = "reverse words of a sentence";
  String result = reverseWords(str);
  System.out.println(result);
 }

 private static String reverseWords(String str)
 {
  char[] chars = str.toCharArray();
  reverse(chars, 0, chars.length - 1);
  int wordStart = 0;
  int wordEnd = 0;
  while (wordEnd < chars.length)
  {
   if (chars[wordEnd] == ' ')
   {
    reverse(chars, wordStart, wordEnd - 1);
    wordStart = wordEnd + 1;
   }
   wordEnd++;
  }
  reverse(chars, wordStart, wordEnd - 1);
  return new String(chars);
 }

 private static void reverse(char[] chars, int i, int j)
 {
  while (i < j)
  {
   char temp = chars[i];
   chars[i] = chars[j];
   chars[j] = temp;
   i++;
   j--;
  }
 }
}
```

Rotate an array k times to left

Problem

Rotate an array k times to its left.

Solution

Reverse the whole array, Then reverse the part 0 to n-k and n-k+1 to n.

Code

```
public class RotateKTimes
{
 public static void main(String[] args)
 {
  int[] array =
  { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  printArray(array);
  rotateLeftKTimes(array, 7);
  printArray(array);
 }
```

```
 private static void rotateLeftKTimes(int[] array, int k)
 {
  reverse(array, 0, array.length - 1);
  reverse(array, 0, array.length - k - 1);
  reverse(array, array.length - k, array.length - 1);
 }

 private static void reverse(int[] array, int i, int j)
 {
  while (i < j)
  {
   int temp = array[i];
   array[i] = array[j];
   array[j] = temp;
   i++;
   j--;
  }
 }

 private static void printArray(int[] array)
 {
  for (int i : array)
   System.out.print(i + ",");
  System.out.println();

 }

}
```

Merge two sorted arrays in a single array with space

Problem

A sorted array has M elements and N blank spaces. Another array has N sorted elements. merge this two arrays in the first array of size M+N

Solution

Start filling the array from back side. That is from the blank positions of first array. As the space is equal to the second array size, it will never overwrite the first array's original values.

Code

```
public class MergeInSingleArray
{
 public static void main(String[] args)
 {
  int M = 10;
  int N = 12;
  int[] array1 = new int[M + N];
  int[] array2 = new int[N];
  fillElementsSorted(array1, M);
  fillElementsSorted(array2, N);
  printArray(array1);
  printArray(array2);
  merge(array1, array2, M, N);
  printArray(array1);
 }

 private static void printArray(int[] array)
 {
```

```
  for (int i : array)
   System.out.print(i + ",");
  System.out.println();

 }

 private static void merge(int[] array1, int[] array2, int M, int N)
 {
  int index = M + N - 1;
  int i = M - 1;
  int j = N - 1;
  while (true)
  {
   if (array1[i] > array2[j])
    array1[index--] = array1[i--];
   else
    array1[index--] = array2[j--];
   if (j < 0)
    break;
   if (i < 0)
   {
    while (index >= 0)
    {
     array1[index] = array2[index];
     index--;
    }
    break;
   }
  }
 }

 private static void fillElementsSorted(int[] array1, int fillCount)
 {
  array1[0] = (int) (Math.random() * 5);
  for (int i = 1; i < fillCount; ++i)
  {
   array1[i] = array1[i - 1] + (int) (Math.random() * 5);
  }
 }

}
```

Rotate a string to make another

Problem

Given two string find whether one string can be formed by rotating another string.

Solution

Suppose string b is to be formed by rotating string a. We concatenate string a with string a. then find for substring b in that concatenation.

Code

```
public class RotateString
{
 public static void main(String[] args)
 {
  String string1 = "rotation";
  String string2 = "tionrota";
  System.out.println(isRotationPossible(string1, string2));
 }
```

```
 private static boolean isRotationPossible(String string1, String string2)
 {
  String str = string1 + string1;
  return str.contains(string2) && string1.length() == string2.length();
 }

}
```

Maximum product subarray

Problem

Given an integer array with negative numbers and zero find the subarray with maximum product, i.e. find the contiguous array elements that produce maximum product.

Solution

Wherever there is 0 that splits the array into subarrays. We need to find the maximum product of these subarrays individually and return the largest product. Inside this subproblem if there are even number of negative elements then maximum product is the complete product of the array. If there are odd number of negative elements, then make two subarrays once leaving the leftmost negative element and once leaving the rightmost negative element. The maximum of these two products will be returned.

Code

```
public class MaxProductSubArray
{
 public static void main(String[] args)
 {
  int[] arr =
  { 1, 2, -1, 4, 0, 5, -6, -5, -6, 2, 0, 3, -4, 3, -2, 4, -3 };
  int[] returnIndices = new int[2];
  long maxProduct = findMaxProduct(arr, returnIndices);
  System.out.println("Maximum product " + maxProduct);
  System.out.println("Indices " + returnIndices[0] + " - "
    + returnIndices[1]);
 }

 private static long findMaxProduct(int[] arr, int[] returnIndices)
 {
  int startIndex = 0;
  long maxProduct = 0;
  int[] indices = new int[2];
  for (int index = 0; index < arr.length; ++index)
  {
   if (arr[index] == 0 && index >= startIndex)
   {
    long product = findMaxProductWithoutZero(arr, startIndex,
      index - 1, indices);
    if (product > maxProduct)
    {
     maxProduct = product;
     returnIndices[0] = indices[0];
     returnIndices[1] = indices[1];
    }
    startIndex = index + 1;
   } else if (index == arr.length - 1)
   {
    long product = findMaxProductWithoutZero(arr, startIndex, index,
      indices);
    if (product > maxProduct)
    {
     maxProduct = product;
     returnIndices[0] = indices[0];
     returnIndices[1] = indices[1];
    }
```

```
    }
   }
   return maxProduct;
  }

  private static long findMaxProductWithoutZero(int[] arr, int startIndex,
    int endIndex, int[] returnIndices)
  {
   if (startIndex > endIndex || startIndex < 0 || endIndex >= arr.length)
    return 0;
   int negativeCount = 0;
   int firstNegativeIndex = -1;
   int lastNegativeIndex = -1;
   for (int index = startIndex; index <= endIndex; ++index)
   {
    if (arr[index] < 0)
    {
     negativeCount++;
     if (firstNegativeIndex == -1)
      firstNegativeIndex = index;
     lastNegativeIndex = index;
    }
   }
   if (negativeCount % 2 == 0)
    return findMaxProductWithoutNegative(arr, startIndex, endIndex,
      returnIndices);
   else
   {
    int[] indices = new int[2];
    long maxProduct = findMaxProductWithoutNegative(arr,
      firstNegativeIndex + 1, endIndex, indices);
    returnIndices[0] = indices[0];
    returnIndices[1] = indices[1];
    long maxProduct2 = findMaxProductWithoutNegative(arr, startIndex,
      lastNegativeIndex - 1, indices);
    if (maxProduct2 > maxProduct)
    {
     maxProduct = maxProduct2;
     returnIndices[0] = indices[0];
     returnIndices[1] = indices[1];
    }
    return maxProduct;
   }
  }

  private static long findMaxProductWithoutNegative(int[] arr,
    int startIndex, int endIndex, int[] indices)
  {
   if (startIndex > endIndex || startIndex < 0 || endIndex >= arr.length)
    return 0;
   long product = 1;
   for (int index = startIndex; index <= endIndex; ++index)
    product *= arr[index];
   indices[0] = startIndex;
   indices[1] = endIndex;
   return product;
  }
}
```

Maximum sum submatrix

Problem

Given a matrix which contains positive and negative integers. Find the submatrix which has the maximum sum.

Brute force

There are total O(n^2) row range and O(n^2) column range. So total O(n^4) sub matrix. To find the sum of any sub matrix we need to do a O(n^2) operation. So the brute force algorithm will have a complexity of O(n^6).

Better Solution

We will create a same sized matrix to keep the vertical sum of the original matrix. For example verticalSum[i,j]=arr[0,j]+arr[1,j]+...+arr[i,j]. Now we will take the row range and move from left to right to find the maximum sum. There are O(n^2) row range possible. Row range will be a range from rowStart to rowEnd. Both of these variables can vary from 1 to n. So there are O(n^2) rowRange possible. For any starting row we start with one row and move from left to right. Then we take 2 rows and move from left to right. While doing so, we maintain an array sum which maintains the vertical sum of the selected row range. This sum array and vertical sum matrix will help us find the sum for a new sub matrix in O(1) time. So the total time complexity will be O(n^3). O(n^2) for row range, and we will have a O(column) operation inside that. So the total complexity will become O(n^3).

Code

```
public class LargestSumSubMatrix
{
 public static void main(String[] args)
 {
  int[][] arr =
  {
  { 1, -2, -7, 0 },
  { -6, 2, 9, 2 },
  { -4, -2, -1, 4 },
  { -1, -8, 0, -4 } };
  int[] leftRightTopBottom = new int[4];
  int maxsum = findMaximumSumSubMatrix(arr, leftRightTopBottom);
  System.out.println("max sum: " + maxsum);
  System.out.println("indices left right top bottom");
  for (int index : leftRightTopBottom)
   System.out.print(index + ",");
 }

 private static int findMaximumSumSubMatrix(int[][] arr,
   int[] leftTopRightBottom)
 {
  leftTopRightBottom[0] = 0;
  leftTopRightBottom[1] = 0;
  leftTopRightBottom[2] = 0;
  leftTopRightBottom[3] = 0;
  int rows = arr.length;
  int cols = arr[0].length;
  int[] sum = new int[cols];
  int[] pos = new int[cols];
  int localMax;
  int maxSum = arr[0][0];
  int[][] verticalSum = new int[rows][cols];

  for (int iRow = 0; iRow < rows; iRow++)
  {
   for (int jCol = 0; jCol < cols; jCol++)
   {
    if (jCol == 0)
    {
     verticalSum[jCol][iRow] = arr[jCol][iRow];
    } else
    {
     verticalSum[jCol][iRow] = arr[jCol][iRow]
       + verticalSum[jCol - 1][iRow];
    }
   }
  }
```

```
  for (int iRow = 0; iRow < rows; iRow++)
  {
   for (int k = iRow; k < rows; k++)
   {
    for (int index = 0; index < cols; index++)
    {
     sum[index] = 0;
     pos[index] = 0;
    }
    localMax = 0;
    int tmp = 0;
    if (iRow > 0)
    {
     tmp = verticalSum[iRow - 1][0];
    }
    sum[0] = verticalSum[k][0] - tmp;
    for (int j = 1; j < cols; j++)
    {
     tmp = 0;
     if (iRow > 0)
     {
      tmp = verticalSum[iRow - 1][j];
     }
     if (sum[j - 1] > 0)
     {
      sum[j] = sum[j - 1] + verticalSum[k][j] - tmp;
      pos[j] = pos[j - 1];
     } else
     {
      sum[j] = verticalSum[k][j] - tmp;
      pos[j] = j;
     }
     if (sum[j] > sum[localMax])
     {
      localMax = j;
     }
    }
    if (sum[localMax] > maxSum)
    {
     maxSum = sum[localMax];
     leftTopRightBottom[0] = pos[localMax];
     leftTopRightBottom[1] = localMax;
     leftTopRightBottom[2] = iRow;
     leftTopRightBottom[3] = k;
    }
   }
  }
  return maxSum;
 }
}
```

Expand the array

Problem

You are given a character array like this **a3b1c1d1e4f0g11.** You will have to expand the array by repeating the characters denoted by the following numbers. For example the above character array will be expanded to **aaabcdeeeegggggggggg**. The given array will have more than enough trailing spaces such that you can modify the array in place.

Solution

We will solve this by recursion. Our first reading location and writing location is 0. As we find a3 we know that this will take 3 spaces. But if we write it now it will override b. So we know from a3 that next write will begin from 4th position and reading will start from 3rd position. So we call this recursive function to read from 3rd position and write it from 4th position. Then after the recursive function returns, we write the first 3 positions. The recursion ends when we find space character in the reading position.

Code

```
public class ExpandArray
{
 public static void main(String[] args)
 {
  char[] arr = "a2b1c1d1e4f0g11                 ".toCharArray();
  expand(arr);
  for (char ch : arr)
   System.out.print(ch);
 }

 private static void expand(char[] arr)
 {
  expand(arr, 0, 0);
 }

 private static void expand(char[] arr, int startReading, int startWriting)
 {
  char ch = arr[startReading++];
  if (ch == ' ')
   return;
  int count = 0;
  while (Character.isDigit(arr[startReading]))
  {
   count = count * 10 + arr[startReading] - 48;
   startReading++;
  }
  expand(arr, startReading, startWriting + count);
  for (int i = 0; i < count; ++i)
   arr[startWriting + i] = ch;
 }
}
```

Sort to bring anagrams closer

Problem

Given an array of strings sort it such that the anagrams come side by side. A word is called anagram of another word, if one can be formed by rearranging the letters of the other, without any addition or deletion of letters.

Solution

As two anagrams have exact same set of letters and same count of letters, if we sort the letters of two anagrams they will be exactly the same. Now if we use this sorted sequence of a word as its comparison key in sorting, then basically two words having same sorted sequence will stay side by side. We use this technique to get this solution. We create a container class for the words and use the sorted sequence of that word as comparator of that class. Now we sort the array of containers. Then from the sorted containers we recreate the array of strings in the same order.

Code

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
```

```
import java.util.List;


public class AnagramSort
{
 public static void main(String[] args)
 {
  String[]arr={"dog","listen","tip","enlist","pit","god","man","top","pot"};
  sort(arr);
  for(String str:arr)
  {
   System.out.println(str);
  }
 }

 private static void sort(String[] arr)
 {
  List list=new ArrayList();
  for(String str:arr)
  {
   list.add(new Anagram(str));
  }
  Collections.sort(list);
  for(int i=0;i < arr.length;++i)
  {
   arr[i]=list.get(i).str;
  }
 }
 public static class Anagram implements Comparable
 {
  String str;
  public Anagram(String str)
  {
   this.str=str;
  }
  public String getSortedString()
  {
   char[]arr=str.toCharArray();
   Arrays.sort(arr);
   return new String(arr);
  }
  @Override
  public int compareTo(Anagram o)
  {
   return getSortedString().compareTo(o.getSortedString());
  }
 }
}
```

Two missing numbers

Problem

You are given an array of natural numbers from 1 to n in an array where two numbers are missing. So the given array size is n-2. You will have to find the two missing numbers.

Solution

We can use a hash map or marker array of size n to solve this problem in O(n) time. But that would use O(n) extra space. We can solve this problem in O(n) time and using constant extra space.

We will calculate the sum of n-2 numbers and sum of the n-2 numbers squared in one iteration. As we know the sum of all numbers from 1 to n is n*(n+1) / 2. And the sum of all numbers squared from 1 to n is n*(n+1)*(2n+1) / 6. So we can find the sum and squared

sum of the missing number by subtracting it from total sum and squared sum of n numbers. So now we know a+b and a^2+b^2. From this we can calculate a and b, the two missing numbers.

Code

```
import java.util.ArrayList;
import java.util.List;

public class TwoMisingNumbers
{
 public static void main(String[] args)
 {
  List list = new ArrayList();
  for (int i = 1; i < 100; ++i)
   list.add(i);
  list.remove(new Integer(12));
  list.remove(new Integer(79));
  Integer[]missing=getTwoMissingNumbers(list);
  System.out.println(missing[0]+","+missing[1]);
 }

 private static Integer[] getTwoMissingNumbers(List list)
 {
  int n=list.size()+2;
  int expectedSum=n*(n+1)/2;
  int expectedSquaredSum=n*(n+1)*(2*n+1)/6;
  int sum=0;
  int squaredSum=0;
  for(int num:list)
  {
   sum+=num;
   squaredSum+=num*num;
  }
  int xplusy=expectedSum-sum;
  int xsquareplusysquare =expectedSquaredSum-squaredSum;
  int twoxy=xplusy*xplusy-xsquareplusysquare;
  int xminusy=(int)Math.sqrt(xsquareplusysquare-twoxy);
  int x=(xplusy+xminusy)/2;
  int y=(xplusy-xminusy)/2;
  return new Integer[]{x,y};
 }


}
```

Maximize stock profit simple

Problem

You are given the prices of a given stock for some days. You were allowed to own at most one stock at any time and you must not hold any stock at the end of that period. Find out the maximum profit that you could have.

Solution

The strategy is to sell the stock at the end of every increasing sequence. And buy at the end of a decreasing sequence. So if at any day the stock price is greater than the previous day, the stock profits increase by the amount of that difference.

Code

```
public class StockPriceMaxProfitSimple {
 public static void main(String[] args) {
  int[] prices = { 400, 402, 435, 398, 378, 400, 432, 432, 402 };
  System.out.println(getMaxProfit(prices, false));
 }
```

```
 private static int getMaxProfit(int[] prices, boolean b) {
  if (prices.length == 0)
   return 0;
  int previousPrice = prices[0];
  int profit = 0;
  for (int i = 0; i < prices.length; ++i) {
   if (prices[i] > previousPrice) {
    profit += (prices[i] - previousPrice);
   }
   previousPrice = prices[i];
  }
  return profit;
 }
}
```

Sum of array except current element

Problem

Replace an array element a(i) with sum(a)-a(i) without using the '-' operator. Where sum(a)=a(0)+a(1)+...+a(n)

Solution

First we will add the numbers from left to right and place it to a new array in such a way so that each element of the new array e(i)=a(0)+a(1)+...+a(i-1). Then we will do the same thing from right to left. In this way the ith element will include all the elements from its left and right but not the element a(i). As we are always doing a sum we don't need a '-' operator.

Code

```
package dsalgo;

public class ArraySumExceptCurrent {

 public static void main(String[] args) {
  int[]arr={3,1,4,5,3,4,12,3};
  int[]result=new int[arr.length];
  int i=0;
  int sum=0;
  while(i < arr.length){
   int temp=arr[i];
   result[i++]=sum;
   sum+=temp;
  }
  i--;
  sum=0;
  while(i>=0){
   int temp=arr[i];
   result[i--]+=sum;
   sum+=temp;
  }
  for(int j=0; j < result.length; ++j){
   System.out.print(result[j]+", ");
  }
 }

}
```

Maximum arithmetic sequence in an array

Problem

You are given an array of integers. Find the length of the longest arithmetic sequence in the array. An arithmetic sequence is contiguous array elements which are in arithmetic progression.

Solution

We iterate over the array and find the difference between the consecutive elements and keep track of the longest running count of same difference. If current difference is different than the previous difference then we reset the count. If the length of the longest running difference is k. Then the longest arithmetic sequence is of length k+1.

Code

```
package com.dsalgo;

public class MaxArithmeticSequence {

 public static void main(String[] args) {
  int[]arr={2,5,3,6,9,12,15,34,23};
  findMaxArithmeticSequence(arr);
 }

 private static void findMaxArithmeticSequence(int[] arr) {
  int maxLength=0;
  int currentLength=0;
  int prevDiff=0;
  for(int i=1;i < arr.length;++i)
  {
   if(i==1)
   {
    prevDiff=arr[i]-arr[0];
    currentLength=maxLength=2;
    continue;
   }
   if(prevDiff==arr[i]-arr[i-1])
   {
    currentLength++;
    if(currentLength > maxLength)
     maxLength=currentLength;
   }
   else
   {
    currentLength=2;
    prevDiff=arr[i]-arr[i-1];
   }
  }
  System.out.println("max arithmetic sequence length = "+maxLength);
 }

}
```

Reverse linked list iterative

Problem

Reverse a linked list

Solution

Keep track of the previous node as you move forward. link that node in next iteration. This code is self explanatory.

Code

```
public class ReverseLinkedListIterative
{
 public static void main(String[] args)
 {
```

```
  Node a=new Node(4);
  Node b=new Node(6);
  Node c=new Node(2);
  Node d=new Node(9);
  Node e=new Node(5);
  Node f=new Node(3);
  Node g=new Node(6);
  Node h=new Node(2);
  a.next=b;
  b.next=c;
  c.next=d;
  d.next=e;
  e.next=f;
  f.next=g;
  g.next=h;
  a.printLinkedList();
  Node head=reverseLinkedList(a);
  head.printLinkedList();
 }
 public static Node reverseLinkedList(Node head)
 {
  Node prev=null;
  Node next=null;
  Node current=head;
  while(current!=null)
  {
   next=current.next;
   current.next=prev;
   prev=current;
   current=next;
  }
  return prev;
 }
 static class Node
 {
  public Node next;
  public int value;

  public Node(int value)
  {
   this.value = value;
  }
  public void printLinkedList()
  {
   Node head=this;
   while (head != null)
   {
    System.out.print(head.value + "->");
    head = head.next;
   }
   System.out.println();
  }

 }
}
```

Reverse linked list recursive

Problem

Reverse a linked list

Solution

We will solve this by recursion. We will reverse the first link and leave the rest of the reversing on the recursion.

Code

```
public class ReverseLinkedListRecursive
{
 public static void main(String[] args)
 {
  Node a=new Node(4);
  Node b=new Node(6);
  Node c=new Node(2);
  Node d=new Node(9);
  Node e=new Node(5);
  Node f=new Node(3);
  Node g=new Node(6);
  Node h=new Node(2);
```

```
  a.next=b;
  b.next=c;
  c.next=d;
  d.next=e;
  e.next=f;
  f.next=g;
  g.next=h;
  a.printLinkedList();
  Node head=reverseLinkedList(a);
  head.printLinkedList();
 }
 public static Node reverseLinkedList(Node head)
 {
  if(head.next==null)
   return head;
  Node newHead=reverseLinkedList(head.next);
  head.next.next=head;
  head.next=null;
  return newHead;
 }
 static class Node
 {
  public Node next;
  public int value;

  public Node(int value)
  {
   this.value = value;
  }
  public void printLinkedList()
  {
   Node head=this;
   while (head != null)
   {
    System.out.print(head.value + "->");
    head = head.next;
   }
   System.out.println();
  }

 }
}
```

Fold a linked list

Problem

Fold a linked list such that the last element becomes second element, last but one element becomes 4 th element and so on. For example input linked list: 1->2->3->4->5->6->7->8->9-> output linked list 1->9->2->8->3->7->4->6->5->

Solution

Find the middle of the linked list. You can do it by slow and fast pointer approach. Start two pointers from head. Advance one pointer at a rate of one node per iteration. Let's call it slow pointer. Advance another pointer at a rate of two nodes per iteration. Let's call it fast pointer. When the fast pointer will reach the end of the linked list, the slow pointer will be at the middle of the linked list. After finding the middle node, we reverse the right half. then we do a in place merge of the two halves of the linked list.

Code

```
public class FoldLinkedList
{
 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(6);
  Node g = new Node(7);
  Node h = new Node(8);
  Node i = new Node(9);
  a.next = b;
  b.next = c;
  c.next = d;
  d.next = e;
  e.next = f;
  f.next = g;
  g.next = h;
  h.next = i;
  a.printLinkedList();
```

```
  foldLinkedList(a);
  a.printLinkedList();

 }

 public static void foldLinkedList(Node head)
 {
  Node slowPointer = head;
  Node fastPointer = head;
  while (fastPointer != null)
  {
   slowPointer = slowPointer.next;
   fastPointer = fastPointer.next;
   if (fastPointer != null)
    fastPointer = fastPointer.next;
  }
  Node middlePointer = slowPointer;
  Node reverseLastHalf = reverseLinkedList(slowPointer);
  while (reverseLastHalf != null && head != middlePointer)
  {
   Node tempHead = head.next;
   Node tempReverse = reverseLastHalf.next;
   reverseLastHalf.next = head.next;
   head.next = reverseLastHalf;
   head = tempHead;
   reverseLastHalf = tempReverse;
  }
  if (reverseLastHalf != null)
   reverseLastHalf.next = null;
  else
   head.next = null;
 }

 public static Node reverseLinkedList(Node head)
 {
  if (head.next == null)
   return head;
  Node newHead = reverseLinkedList(head.next);
  head.next.next = head;
  head.next = null;
  return newHead;
 }

 static class Node
 {
  public Node next;
  public int value;

  public Node(int value)
  {
   this.value = value;
  }

  public void printLinkedList()
  {
   Node head = this;
   while (head != null)
   {
    System.out.print(head.value + "->");
    head = head.next;
   }
   System.out.println();
  }

 }

}
```

Reverse k nodes in linked list

Problem

Reverse every k nodes in a linked list and return the head appropriately.

Solution

We will use a reverse function which reverses k nodes from head and return kth node as head. We will use this function recursively to reverse every k nodes.

Code

```
public class ReverseKNodes
{
 public static void main(String[] args)
 {
  int[] arr =
  { 4, 5, 3, 6, 8, 3, 5, 7, 3, 7, 9, 4, 6 };
  Node head = createLinkedList(arr);
  printLinkedList(head);
  head = reverse(head, 4);
  printLinkedList(head);
 }

 static Node reverse(Node head, int k)
 {
  Node current = head;
  Node next = null;
  Node prev = null;
  int count = 0;

  while (current != null && count < k)
  {
   next = current.next;
   current.next = prev;
   prev = current;
   current = next;
   count++;
  }

  if (next != null)
  {
   head.next = reverse(next, k);
  }
  return prev;
 }

 private static void printLinkedList(Node head)
 {
  while (head != null)
  {
   System.out.print(head.value + "->");
   head = head.next;
  }
  System.out.println();
 }

 private static Node createLinkedList(int[] arr)
 {
  Node head = null;
  Node current = null;
  for (int element : arr)
  {
   if (head == null)
   {
    head = new Node(element);
    current = head;
   } else
   {
    current.next = new Node(element);
    current = current.next;
   }
  }
  return head;
 }

 static class Node
 {
  public Node next;
  public int value;

  public Node(int value)
  {
   this.value = value;
  }
 }

}
```

Find loop in linked list

Problem

Given a linked list find whether the list is looped or not. A linked list is looped when one of the nodes point to any one of its previous nodes or itself. Also find the length of the loop and the starting point of the loop.

Solution

We take two pointers, one slow, which moves one node at a time and another fast, which moves two nodes at a time. Both the pointers start from head. If the linked list is not looped. fast pointer will hit null pointer. If they meet, that means the linked list is looped. If it is looped then both the pointers are at some point in the loop. We keep one fixed and move another pointer in the loop till both of them meet again. In this way we will find the length of the loop. Now we take both the pointer to the head and move the fast pointer 'loop length' ahead of the slow pointer. Then we again move them one node at a time till they meet. As the fast pointer is 'loop length' ahead of the slow pointer, when slow pointer reaches starting of the loop fast will also reach there after completing one loop. So when these two pointers meet that point will be the starting point of the loop.

Code

```
public class LinkedListWithLoop
{
 public static void main(String[] args)
 {
  Node head = new Node(1);
  Node middle = head.append(2).append(3).append(4).append(5);
  Node tail = middle.append(6).append(7).append(8).append(9).append(10)
    .append(11);
  tail.next = middle;
  findLoopInformation(head);
 }

 private static void findLoopInformation(Node head)
 {
  Node slowPointer = head;
  Node fastPointer = head;
  boolean isLooped = false;
  while (fastPointer != null && fastPointer.next != null)
  {
   slowPointer = slowPointer.next;
   fastPointer = fastPointer.next.next;
   if (slowPointer == fastPointer)
   {
    isLooped = true;
    break;
   }
  }
  if (isLooped)
  {
   System.out.println("Linkedlist is looped");
   fastPointer = fastPointer.next;
   int count = 1;
   while (slowPointer != fastPointer)
   {
    fastPointer = fastPointer.next;
    count++;
   }
   System.out.println("Loop length = " + count);
   slowPointer = head;
   fastPointer = head;
   while (count-- > 0)
   {
    fastPointer = fastPointer.next;
   }
   while (slowPointer != fastPointer)
```

```
   {
    slowPointer = slowPointer.next;
    fastPointer = fastPointer.next;
   }
   System.out.println("Loop starting point = " + slowPointer.value);
  } else
  {
   System.out.println("Linkedlist is NOT looped");
  }
 }

 private static class Node
 {
  public Node next;
  public int value;

  public Node(int value)
  {
   this.value = value;
  }

  public Node append(int value)
  {
   Node node = new Node(value);
   next = node;
   return node;
  }
 }
}
```

Linked list Y shape

Problem

Given two linked lists find out whether they are converged to a single linked list or not, if yes, find there point of convergence.

Solution

If the linked lists are convergent, then their last node will be same. So we traverse along both the nodes and find the last node of each linked lists. If they are equal then they are convergent.
To find the point of convergence, we first find their corresponding lengths and find their difference. Then we traverse the longer linked list number of nodes equal to the difference. After that if we traverse along the two nodes simultaneously when we find a common node that is the node of convergence.

Code

```
public class LinkedListYShape
{

 public static void main(String[] args)
 {
  Node head1 = new Node(1);
  Node middle1 = head1.append(2).append(3).append(4);
  middle1.append(6).append(7).append(8).append(9);
  Node head2 = new Node(10);
  Node middle2 = head2.append(3).append(5).append(6).append(11)
    .append(24);
  middle2.next = middle1;
  findIntersection(head1, head2);

 }
```

```
private static void findIntersection(Node head1, Node head2)
{
 int count1 = 0;
 int count2 = 0;
 Node ptr1 = head1;
 Node ptr2 = head2;

 while (true)
 {
  if (ptr1.next == null)
   break;
  ptr1 = ptr1.next;
  count1++;
 }
 while (true)
 {
  if (ptr2.next == null)
   break;
  ptr2 = ptr2.next;
  count2++;
 }
 boolean isMerged = ptr1 == ptr2;
 if (isMerged)
 {
  System.out.println("The linked lists are merged");
  Node longer = head1;
  Node shorter = head2;
  if (count1 < count2)
  {
   longer = head2;
   shorter = head1;
  }
  int diff = Math.abs(count1 - count2);
  while (diff-- > 0)
  {
   longer = longer.next;
  }
  while (longer != shorter)
  {
   longer = longer.next;
   shorter = shorter.next;
  }
  System.out.println("Common Node = " + longer.value);
 } else
 {
  System.out.println("The linked lists are NOT merged");
 }
}

private static class Node
{
 public Node next;
 public int value;

 public Node(int value)
 {
  this.value = value;
 }

 public Node append(int value)
 {
  Node node = new Node(value);
  next = node;
  return node;
 }
}
```

```
}
```

Find kth node from end in linked list

Problem

Given a linked list find the kth node from end.

Solution

Start a pointer from head and move it k nodes. Then take another pointer from head and move them simultaneously. When the front pointer reaches end, next one will reach k nodes from end.

Code

```
public class LinkedListKthElementFromEnd
{
 public static void main(String[] args)
 {
  Node head = new Node(1);
  head.append(2).append(3).append(4).append(5).append(6).append(7)
     .append(8).append(9);
  Node result = findKFromEnd(head, 3);
  System.out.println(result.value);
 }

 private static Node findKFromEnd(Node head, int k)
 {
  Node ahead = head;
  while (k-- > 0)
   ahead = ahead.next;
  while (ahead != null)
  {
   head = head.next;
   ahead = ahead.next;
  }
  return head;
 }

 private static class Node
 {
  public Node next;
  public int value;

  public Node(int value)
  {
   this.value = value;
  }

  public Node append(int value)
  {
   Node node = new Node(value);
   next = node;
   return node;
  }
 }

}
```

Get Find and Delete all O(1)

Problem

You will have to design a telephone directory where the following operations would be supported.
1. GET: It will provide a number which is not assigned to anyone
2. CHECK: Given a number it will tell whether it is assigned to anyone or not
3. RELEASE: It will recycle or release a number

Normal solutions

We can implement an array of numbers, then
GET will have O(n), because we need to find through the list for available numbers
CHECK will have O(1)
RELEASE will have O(1)

If we implement via linked list of available numbers, then
GET will have O(1) as we can return the head
CHECK will have O(n) as we will have to iterate through the linked list
RELEASE will have O(1) as we can just add it at the head.

Solution

We can have all 3 operations in O(1) by a combination of array and linked list. every number is represented by a doubly linked node in its corresponding array location and a doubly linked list runs through the nodes in the array.
GET will have O(1) as we can remove the head of the linked list in O(1) time.
CHECK will have O(1) as we can directly go to the array index, as every index contain the corresponding number's node. Then we can check the availability in the node.
RELEASE will have O(1) as we can go to the node directly by array index and then add it to the doubly linked list. We can add the node to the head in O(1) time.

In this arrangement we can also check the availability of a given number and get that particular number as well. For example if somebody wants a beautiful or lucky number from the directory.

Code

```
public class TelephoneDirectory
{
 public static void main(String[] args)
 {
  Directory directory = new Directory(5);
  int number1 = directory.getAvailableNumber();
  System.out.println(number1);
  int number2 = directory.getAvailableNumber();
  System.out.println(number2);
  System.out.println(directory.isAvailable(3));
  int number3 = directory.getAvailableNumber();
  System.out.println(number3);
  System.out.println(directory.isAvailable(3));
  int number4 = directory.getAvailableNumber();
  System.out.println(directory.isAvailable(3));
  System.out.println(number4);
  int number5 = directory.getAvailableNumber();
  System.out.println(number5);
  int number6 = directory.getAvailableNumber();
  System.out.println(number6);
  int number7 = directory.getAvailableNumber();
  System.out.println(number7);
  directory.release(3);
  System.out.println(directory.isAvailable(3));
  int number8 = directory.getAvailableNumber();
  System.out.println(number8);
  System.out.println(directory.isAvailable(3));
 }

 private static class Directory
 {
```

```
Node[] nodes;
Node head;

public Directory(int maxNumbers)
{
 nodes = new Node[maxNumbers];
 for (int i = 0; i < maxNumbers; ++i)
 {
  nodes[i] = new Node(i);
  if (i != 0)
  {
   nodes[i].previous = nodes[i - 1];
   nodes[i - 1].next = nodes[i];
  }
 }
 nodes[maxNumbers - 1].next = nodes[0];
 nodes[0].previous = nodes[maxNumbers - 1];
 head = nodes[0];
}

public int getAvailableNumber()
{
 if (head == null)
  return -1;
 int temp = head.number;
 head.available = false;
 if (head.next == head)
  head = null;
 else
 {
  head.previous.next = head.next;
  head.next.previous = head.previous;
  head = head.next;
 }
 return temp;
}

public boolean isAvailable(int number)
{
 return nodes[number].available;
}

public void release(int number)
{
 if (nodes[number].available == false)
 {
  nodes[number].available = true;
  if (head == null)
  {
   head = nodes[number];
   nodes[number].next = nodes[number];
   nodes[number].previous = nodes[number];
  } else
  {
   nodes[number].next = head.next;
   nodes[number].previous = head;
   head.next.previous = nodes[number];
   head.next = nodes[number];
  }
 }
}

private static class Node
{
 boolean available = true;
 Node next;
 Node previous;
 int number;
```

```
   public Node(int number)
   {
    this.number = number;
   }
  }
 }

}
```

Calculate power

Problem

Calculate x to the power y.

Constraint

Complexity should be less than O(n).

Solution

Suppose we are trying to calculate x to the power y and y is 4. Then we can do result=1; Then 4 times (result=result*x). In this way the complexity is O(n). In another way, we can do result=x*x; then we can do result =result *result; in this way the number of multiplication is 2 but y=4. So for power of 2s we can clearly see the number of multiplication is log(n) time. What happens for other numbers like when y=7. We can divide seven in 4+2+1. so x^(4+2+1)=(x^4) * (x^2) * (x^1).
So we can see that all powers are now divided into power of some numbers which are powers of 2. For example x^y where y is 13, we can do 8+4+1. Clearly we can see when y is represented in binary, the set bits are indicative of how to calculate the result. So whenever the bit is set multiply the powers to the result. and then do power=power*power for the next bit.

Code

```
public class CalculatePower
{
 public static void main(String[] args)
 {
  int x = 4;
  int y = 10;
  long power = x;
  long answer = 1;
  while (y != 0)
  {
   if ((y & 1) == 1)
   {
    answer *= power;
   }
   y >>= 1;
   power *= power;
  }
  System.out.println(answer);
 }
}
```

Next power of two

Problem

Given a number find the lowest number which is power of 2 and greater than the given number.

Solution

We will keep on shifting the bits to right till the number becomes zero. So if the last set bit of the number is at k distance from the right side, we will have to shift the number k times to make it zero. During every right shift we will left shift a result which has initial value 1. Then the result will have (k+1)th bit set after the iteration and all other bits set to zero. Any number where exactly one bit is set is a power of two. And the result will be greater than the given number as the highest set bit is in the left side. The next smaller power of two would have it's kth bit set, which will be either equal or smaller than the previous number. So this result is the lowest number which is greater than the given number and a power of 2.

Code

```
public class NextPowerOfTwo
{
 public static void main(String[] args)
 {
  long num = 128;
```

```
  long result = findNextPowerOfTwo(num);
  System.out.println(result);
 }

 private static long findNextPowerOfTwo(long num)
 {
  long result = 1;
  while (num != 0)
  {
   num >>= 1;
   result <<= 1;
  }
  return result;
 }
}
```

Count number of ones till N in binary

Problem

Given a number n find total number of ones in the binary representation of the numbers 1, 2, 3, 4, ..., n

For example, if n=5

1=1 (number of ones=1)
2=10 (number of ones=1)
3=11 (number of ones=2)
4=100 (number of ones=1)
5=101 (number of ones=2)

output
7 (1+1+2+1+2)

Solution

Initialize a count to 1 for odd numbers and 0 for even numbers. If the bth bit is set then add to a count 2^(b-1)*b+n%2^b+1;

Code

```
public class NumberOfOnes
{

 /**
  * given n find the total number of ones in the binary representation of
  * numbers 1,2,3,...,n
  *
  * @param args
  */
 public static void main(String[] args)
 {
  System.out.println(getNumberOfOnes(30));
 }

 public static int getNumberOfOnes(int num)
 {
  int p = 1, cnt = 0;

  if ((num & 1) != 0)
   cnt++;
```

```
  while (1 << (p - 1) < num)
  {
   if ((num & (1 << p)) != 0)
    cnt += (1 << (p - 1)) * (p) + num % (1 << p) + 1;
   p++;
  }
  return cnt;
 }

}
```

Swap without temp

Problem

Swap two integers without using a temp variable.

Solution

We will use some excellent property of XOR to solve this. If we xor two numbers then the result will give you back the original numbers when again xor ed with the other number. So if ANS=A XOR B
then ANS XOR A = B and ANS XOR B = A.

Code

```
public class SwapWithoutTemp
{
 public static void main(String[] args)
 {
  int a = 5;
  int b = 6;
  a = a ^ b;
  b = a ^ b;
  a = a ^ b;
  System.out.println(a);
  System.out.println(b);
 }
}
```

Array next element

Problem

Given an integer array replace each number with its next minimum number in the array. For example if input array is {3,4,2,6,5,1,8,4} output would be{2,2,1,5,1,0,4,0}

Brute force

Create a new array of same size. For each element start scanning the element from its immediate next index, till you first find a smaller element. In this way the complexity is O(n^2) and extra space of size n is used.

Better Solution

We will scan the array from right side to left side. We will maintain a stack to keep track of the next smaller elements. Starting from right we will take a number and try to put that number in the stack. We will always put the number on smaller numbers. So as long as stack top element is bigger than the given number we will keep popping the stack. At some point either a smaller element will appear or stack will become empty. At this point we will replace the array element with the top element of the stack or put zero if the stack is empty. This way we will try to put the number on stack then replace the number in the array and keep on moving from right to left. the complexity of this approach will be O(n). We can prove it by saying that we will only handle one number once in the array and the stack can not have more than n pop or n push.

Code

```
import java.util.LinkedList;

public class NextSmallInteger
{

 /**
  * given an array form another array where each element of previous
```

```
  * array is replaced with its next minimum number in the array
  * @param args
  */
 public static void main(String[] args)
 {
  int[] input={3,4,5,2,7,5,7,3,8,2,5,7,9,1,3};
  int[]output=new int[input.length];
  LinkedList<Integer> stack=new LinkedList<Integer>();
  for(int i=input.length-1;i>=0;--i)
  {
   int currentNum=input[i];
   if(stack.peek()==null)
   {
    output[i]=0;
    stack.push(currentNum);
    continue;
   }
   while(stack.size()!=0 && stack.peek()>=currentNum)
   {
    stack.pop();
   }
   output[i]=stack.peek()==null?0:stack.peek();
   stack.push(currentNum);
  }
  for(int i=0;i < output.length;++i)
  {
   System.out.print(output[i]+",");
  }
 }

}
```

Stack with get minimum

Problem

Implement a stack which provides an efficient get minimum function along with regular push and pop functionality.

Brute force

In normal implementation of the stack with array or linked list the push and pop has O(1) complexity, But the brute force implementation of get minimum will result in a O(n) complexity.

Solution

We can implement a O(1) complexity for push, pop and get minimum by using O(n) extra space. In this solution we keep another stack for minimum calculation. Whenever a new number is inserted if it is smaller than the stack top it is inserted into this special stack otherwise the minimum is again inserted into the special stack. When get minimum is called, top of minimum stack is returned. When a number is popped from the stack, one number is popped from the min stack also. This can be done in other way also, if the number being inserted is bigger than the min stack top, nothing is done on the min stack. when a number is popped min stack is popped only when its top is equal to the number. But the former one is easy to code and both has same complexity.

Code

```
import java.util.Stack;

public class StackMinimum
{
 private Stack<Integer> stack = new Stack<Integer>();
 private Stack<Integer> minStack = new Stack<Integer>();

 public Integer push(Integer item)
 {
  if(stack.empty())
  {
   stack.push(item);
   minStack.push(item);
   return item;
  }
  Integer currentMin=minStack.peek();
  if(currentMin < item)
   minStack.push(currentMin);
  else
   minStack.push(item);
  stack.push(item);
  return item;
 }

 public Integer pop()
 {
  if(stack.size()==0)
   return null;
  minStack.pop();
  return stack.pop();
 }

 public Integer getMinimum()
```

```
 {
  if(minStack.empty())
   return null;
  return minStack.peek();
 }

 public int size()
 {
  return stack.size();
 }

}
```

Sort a stack

Problem

A stack is given which supports the following methods. Push, pop, peek and isEmpty. Sort this stack irrespective of its inner structure.

Solution

We will recursively pop the stack till it is empty, Then we will try to put the popped number with another recursive function named insertSorted. This will push to the stack if the top of the stack is greater than the number, if it is not, then it will pop the number and call the insertSorted with the popped number and the rest of the stack. After it returns it will push the number in correct place.

Code

```
import java.util.Stack;

public class SortStack
{
 public static void main(String[] args)
 {
  Stackstack=new Stack();
  stack.push(5);
  stack.push(3);
  stack.push(9);
  stack.push(2);
  stack.push(6);
  sort(stack);
  while(!stack.isEmpty())
  {
   System.out.println(stack.pop());
  }
 }

 static void sort(Stack stack)
 {
  if (stack.isEmpty())
   return;
  Integer top = stack.pop();
  sort(stack);
  insertSorted(top, stack);
  return;
 }

 static void insertSorted(Integer top, Stack stack)
 {
  if (stack.isEmpty() || stack.peek() > top)
  {
   stack.push(top);
   return;
  }
  Integer smaller = stack.pop();
  insertSorted(top, stack);
  stack.push(smaller);
 }
```

```
}
```

Find all permutations

Problem

Find all the permutations of a given string whose letters are unique.

Solution

Suppose the string is abcd. Then in its permutations, first letter can be a, b, c or d. If we decide any one of these as the first letter. Then the subproblem arises of having permutations of the rest of the letters. We solve this by recursion.

Code

```
import java.util.ArrayList;
import java.util.List;

public class FindPermutations
{
 public static void main(String[] args)
 {
  findAllPermutations("abcde");
 }

 private static void findAllPermutations(String string)
 {
  Listcharacters=new ArrayList();
  for(char ch:string.toCharArray())
   characters.add(ch);
  findAllPermutations(new ArrayList(),characters);
 }

 private static void findAllPermutations(List prefix,
   List suffix)
 {
  if(suffix.size()==1)
  {
   for(Character ch:prefix)
    System.out.print(ch);
   System.out.println(suffix.get(0));
   return;
  }
  for(int i=0;i < suffix.size();++i)
  {
   Character ch=suffix.get(i);
   prefix.add(ch);
   suffix.remove(i);
   findAllPermutations(prefix,suffix);
   suffix.add(i,ch);
   prefix.remove(ch);
  }
 }


}
```

Find all possible paths in a maze

Problem

Given a maze where some of the cells are blocked, where left top cell is the entry point and right bottom cell is the exit point, find all possible paths from entry to exit which goes through the non blocked cells.

Solution

For finding all possible paths we do depth first traversal. We mark the start point as visited and then recursively try to find all possible paths form the accessible cells from the first cell. For example suppose we are at some arbitrary cell. We first mark the node, then call the function recursively to all its neighboring cells which are not blocked or marked earlier. If some paths are found from these recursive calls. It prepends current cell to those paths and return the result and unmark the current cell.

Code

```
import java.util.ArrayList;
import java.util.List;

public class PathInMaze
{
 public static void main(String[] args)
 {
  boolean[][] maze = new boolean[10][10];
  makeRandomMaze(maze);
  printMaze(maze);
  List> paths = findPaths(maze);
  if (paths == null)
  {
   System.out.println("No path possible");
   return;
  }
  for (List path : paths)
  {
   for (Cell cell : path)
    System.out.print(cell + ",");
   System.out.println();
  }
 }

 private static List> findPaths(boolean[][] maze)
 {
  Cell start = new Cell(0, 0);
  Cell end = new Cell(maze.length - 1, maze[0].length - 1);
  List> paths = findPaths(maze, start, end);
  return paths;
 }

 private static List> findPaths(boolean[][] maze, Cell start,
   Cell end)
 {
  List> result = new ArrayList>();
  int rows = maze.length;
  int cols = maze[0].length;
  if (start.row < 0 || start.col < 0)
   return null;
  if (start.row == rows || start.col == cols)
   return null;
  if (maze[start.row][start.col] == true)
   return null;
  if (start.equals(end))
  {
   List path = new ArrayList();
   path.add(start);
   result.add(path);
   return result;
```

```
 }
 maze[start.row][start.col] = true;
 Cell[] nextCells = new Cell[4];
 nextCells[0] = new Cell(start.row + 1, start.col);
 nextCells[2] = new Cell(start.row, start.col + 1);
 nextCells[1] = new Cell(start.row - 1, start.col);
 nextCells[3] = new Cell(start.row, start.col - 1);

 for (Cell nextCell : nextCells)
 {
  List> paths = findPaths(maze, nextCell, end);
  if (paths != null)
  {
   for (List path : paths)
    path.add(0, start);
   result.addAll(paths);
  }
 }
 maze[start.row][start.col] = false;
 if (result.size() == 0)
  return null;
 return result;
}

private static class Cell
{
 public int row;
 public int col;

 public Cell(int row, int column)
 {
  this.row = row;
  this.col = column;
 }

 @Override
 public boolean equals(Object obj)
 {
  if (this == obj)
   return true;
  if ((obj == null) || (obj.getClass() != this.getClass()))
   return false;
  Cell cell = (Cell) obj;
  if (row == cell.row && col == cell.col)
   return true;
  return false;
 }

 @Override
 public String toString()
 {
  return "(" + row + "," + col + ")";
 }
}

private static void printMaze(boolean[][] maze)
{
 for (int i = 0; i < maze.length; ++i)
 {
  for (int j = 0; j < maze[i].length; ++j)
  {
   if (maze[i][j])
    System.out.print("#|");
   else
    System.out.print("_|");
  }
  System.out.println();
 }
```

```
 }

 private static void makeRandomMaze(boolean[][] maze)
 {
  for (int i = 0; i < maze.length; ++i)
  {
   for (int j = 0; j < maze[0].length; ++j)
   {
    maze[i][j] = (int) (Math.random() * 3) == 1;
   }
  }
  maze[0][0] = false;
  maze[maze.length - 1][maze[0].length - 1] = false;

 }
}
```

Find longest path in a maze

Problem

Given a maze some of whose cells are blocked. Left top is the entry point and right bottom is the exit point. Find the longest possible path from entry to exit that does not contain any blocked cells.

For example,
Input maze

```
_|_|_|#|#|_|#|_|#|
#|#|#|_|#|#|_|#|#|_|
_|_|_|_|#|#|_|_|_|_|
_|_|_|_|#|_|_|#|_|_|
_|_|_|_|#|_|#|_|#|_|
#|_|_|_|_|_|_|#|#|_|
_|#|#|_|#|_|_|#|_|#|
_|#|_|#|#|_|#|_|_|_|
_|_|_|#|#|_|_|#|_|_|
#|#|_|_|#|#|_|_|_|_|
```

Output longest path

(0,0),(0,1),(0,2),(0,3),(1,3),(2,3),(3,3),(4,3),(4,2),(3,2),(2,2),(2,1),(3,1),(3,0),(4,0),(4,1),(5,1),(5,2),(5,3),(5,4),(5,5),(5,6),(6,6),(6,5),
(7,5),(8,5),(8,6),(9,6),(9,7),(9,8),(8,8),(7,8),(7,9),(8,9),(9,9),

Solution

It is solved with the following recursive logic. If we are at any intermediate point along the longest path then from that cell we need to find the longest path to the exit which does not include any of the cells till the current path. From any cell we first mark that cell as blocked. then find all the neighboring cells that are not blocked. We call the recursive function from that neighboring cell to the exit. Find the longest of the paths and return that path after prepending it with the current cell. Then unblock the current cell.

Code

```
import java.util.ArrayList;
import java.util.List;

public class LongestPathInMaze
{
 public static void main(String[] args)
 {
```

```
 boolean[][] maze = new boolean[10][10];
 makeRandomMaze(maze);
 printMaze(maze);
 List< cell > path = findLongestPath(maze);
 if (path == null)
 {
  System.out.println("No path possible");
  return;
 }
 for (Cell cell : path)
  System.out.print(cell + ",");
 System.out.println();
}

private static List< cell > findLongestPath(boolean[][] maze)
{
 Cell start = new Cell(0, 0);
 Cell end = new Cell(maze.length - 1, maze[0].length - 1);
 List< cell > path = findLongestPath(maze, start, end);
 return path;
}

private static List< cell > findLongestPath(boolean[][] maze, Cell start,
  Cell end)
{
 List< cell > result = null;
 int rows = maze.length;
 int cols = maze[0].length;
 if (start.row < 0 || start.col < 0)
  return null;
 if (start.row == rows || start.col == cols)
  return null;
 if (maze[start.row][start.col] == true)
  return null;
 if (start.equals(end))
 {
  List< cell > path = new ArrayList< cell >();
  path.add(start);
  return path;
 }
 maze[start.row][start.col] = true;
 Cell[] nextCells = new Cell[4];
 nextCells[0] = new Cell(start.row + 1, start.col);
 nextCells[2] = new Cell(start.row, start.col + 1);
 nextCells[1] = new Cell(start.row - 1, start.col);
 nextCells[3] = new Cell(start.row, start.col - 1);
 int maxLength = -1;
 for (Cell nextCell : nextCells)
 {
  List< cell > path = findLongestPath(maze, nextCell, end);
  if (path != null && path.size() > maxLength)
  {
   maxLength = path.size();
   path.add(0, start);
   result = path;
  }
 }
 maze[start.row][start.col] = false;
 if (result == null || result.size() == 0)
  return null;
 return result;
}

private static class Cell
{
 public int row;
 public int col;
```

```
  public Cell(int row, int column)
  {
   this.row = row;
   this.col = column;
  }

  @Override
  public boolean equals(Object obj)
  {
   if (this == obj)
    return true;
   if ((obj == null) || (obj.getClass() != this.getClass()))
    return false;
   Cell cell = (Cell) obj;
   if (row == cell.row && col == cell.col)
    return true;
   return false;
  }

  @Override
  public String toString()
  {
   return "(" + row + "," + col + ")";
  }
 }

 private static void printMaze(boolean[][] maze)
 {
  for (int i = 0; i < maze.length; ++i)
  {
   for (int j = 0; j < maze[i].length; ++j)
   {
    if (maze[i][j])
     System.out.print("#|");
    else
     System.out.print("_|");
   }
   System.out.println();
  }
 }

 private static void makeRandomMaze(boolean[][] maze)
 {
  for (int i = 0; i < maze.length; ++i)
  {
   for (int j = 0; j < maze[0].length; ++j)
   {
    maze[i][j] = (int) (Math.random() * 3) == 1;
   }
  }
  maze[0][0] = false;
  maze[maze.length - 1][maze[0].length - 1] = false;

 }
}
```

Towers of Hanoi

Problem

There are 3 rods where discs can be stacked. All the discs are having different diameters. The goal of the game is to move all discs from one tower to another using the third tower. The rules are simple. Only smaller disc can go on top of the bigger discs. So the discs will be sorted according to their sizes and bigger one will be at the bottom. Only one disc can be moved at a time. you must put the removed disc in some tower, it cannot be left in any other places. At the beginning there are some discs in tower one.

The goal is to move them to tower 2 following the above rules.

Solution

Suppose we are moving discs from tower 1 to tower 2 with the help of tower 3. We can see that the bigger disc of the bottom can only be moved when all top discs are moved out of tower 1. As this is the bigger disc it can only go to an empty rod. So this creates a subproblem. To move all the disc from 1 to 2, we need to move n-1 discs from 1 to 3 using 2. Now after moving the top n-1, we can move the bottom one to tower2 as it is empty at that point. After that we need to move all discs from tower 3 to tower 2 with the help of tower 1, as this is currently empty.

Code

```
import java.util.Stack;

public class TowersOfHanoi
{
 public static void main(String[] args)
 {
  Tower towerSource = new Tower("1");
  Tower towerDestination = new Tower("2");
  Tower towerHelper = new Tower("3");
  towerSource.stack.push(4);
  towerSource.stack.push(3);
  towerSource.stack.push(2);
  towerSource.stack.push(1);
  move(towerSource.stack.size(), towerSource, towerDestination,
    towerHelper);
 }

 private static void move(int size, Tower towerSource,
   Tower towerDestination, Tower towerHelper)
 {
  if (towerSource.stack.isEmpty())
   return;
  if (size == 1)
  {
   System.out.println("Move " + towerSource.stack.peek()
     + " from tower " + towerSource.name + " to tower "
     + towerDestination.name);
   towerDestination.stack.push(towerSource.stack.pop());
   return;
  }
  move(size - 1, towerSource, towerHelper, towerDestination);
  move(1, towerSource, towerDestination, towerHelper);
  move(size - 1, towerHelper, towerDestination, towerSource);
 }

 private static class Tower
 {
  public String name;
  public Stack < integer > stack = new Stack < integer >();

  public Tower(String name)
  {
   this.name = name;
  }
 }
}
```

Snakes and ladders

Problem

Given a snakes and ladder board, find out the minimum move to reach to the top.

Brute force

From any cell the dice throw can decide 6 new positions. So if we try to find out all possible paths, it will give 6^n such paths.

Better solution

We can consider any cell of the snakes and ladders board as a vertex. From any vertex we can have 6 different vertex where we can go based on the dice throw. We can think of these as 6 edges with cost 1, going towards 6 different vertexes. We can apply BFS to find the shortest path. We start from cell 1, we mark all the cells that can be reached by 1 throw of dice as 1. if any one of these cells are starting point of a snake or ladder we mark the endpoint of that snake or ladder instead. So we start from cell 1 and after this step all the cells marked as 1 can be reached by 1 throw. Then we put all these nodes in a queue and try to find all the cells that are one throw away from those cells. Then we mark them as 2 and put them in queue. In this way when the cell 100 is reached we found the path. To reconstruct the path we keep the dice throw that caused us to reach a particular cell and the previous cell from where we reach there in two separate arrays.

Code

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Stack;

public class SnakesAndLadders
{

 private static List<Movers> snakesAndLadders;

 public static void main(String[] args)
 {

  snakesAndLadders = new ArrayList<Movers>();

  snakesAndLadders.add(new Movers(9, 31));

  snakesAndLadders.add(new Movers(17, 7));

  snakesAndLadders.add(new Movers(28, 84));

  snakesAndLadders.add(new Movers(85, 41));

  snakesAndLadders.add(new Movers(51, 67));

  snakesAndLadders.add(new Movers(54, 34));

  snakesAndLadders.add(new Movers(62, 19));

  snakesAndLadders.add(new Movers(64, 60));

  snakesAndLadders.add(new Movers(71, 91));

  snakesAndLadders.add(new Movers(80, 99));

  snakesAndLadders.add(new Movers(87, 24));

  snakesAndLadders.add(new Movers(93, 73));


  int[] moveMap = new int[101];

  for (Movers movers : snakesAndLadders)
   moveMap[movers.start] = movers.end;

  int[] minMove = new int[101];
  int[] dice = new int[101];
  int[]fromCell=new int[101];

  LinkedList<Integer> queue = new LinkedList<Integer>();
  queue.add(1);
  boolean finished=false;
  while (!queue.isEmpty()&&!finished)
  {
   int cell = queue.poll();
   for (int i = 1; i <= 6; ++i)
   {
    int newCell = moveMap[cell + i] == 0 ? cell +
      i : moveMap[cell + i];
    if (minMove[newCell] == 0)
    {
     minMove[newCell] = minMove[cell] + 1;
     dice[newCell]=i;
     fromCell[newCell]=cell;
     queue.add(newCell);
    }
    if(newCell==100)
    {
     finished=true;
     break;
    }
```

```
   }

  }
  int cell=100;
  Stack<String> stack=new Stack<String>();
  while(cell!=1)
  {
   stack.push("new cell "+cell);
   stack.push("dice throw "+dice[cell]);

   cell=fromCell[cell];
  }
  while(!stack.isEmpty())
   System.out.println(stack.pop());
 }
}

/**
 * Snakes and Ladders basically do the same thing. They move the
 * coin from one place to another if they move in positive direction
 * we call them ladders otherwise snakes.
 *
 */
class Movers
{

 public int start;

 public int end;

 public boolean goingUp;

 public Movers(int start, int end)
 {

  this.start = start;

  this.end = end;
 }
}
```

Queue using stack

Problem

Implement a queue using stack.

Solution

We can implement a queue using two stacks. while enqueueing we will put into stack one. while dequeuing we will pop from stack two. when stack two is empty we will pop the stack one and put all items into stack two.

Code

```
import java.util.Stack;


public class QueueUsingStack
{
 private Stack<Integer>stackPop=new Stack<Integer>();
 private Stack<Integer>stackPush=new Stack<Integer>();

 public void enqueue(int a)
 {
  stackPush.push(a);
 }
 public int dequeue()
 {
  if(stackPop.isEmpty())
   while(!stackPush.isEmpty())
    stackPop.push(stackPush.pop());
  return stackPop.pop();
 }
}
```

Queue minimum using stack

Problem

Implement a queue which has an efficient get minimum function along with enqueue and dequeue method.

Brute force

when queue is implemented using circular array or linked list, the enqueue and dequeue operation is of O(1) complexity. but the brute force algorithm for implementing the get minimum is of O(n) complexity.

Solution

In our previous example we have seen that we can implement a stack which has push, pop and get minimum method all having O(1) complexity. See StackMinimum. And we have also seen that a queue can be implemented using two stacks. See Queue using stack. Using these two solutions we can implement a queue with get minimum method with O(1) complexity.

Code

```
/*
 * Implement a queue in which push_rear(), pop_front()
 * and get_min() are all constant time operations.
 * first solution using two stacks
 */
public class QueueMinUsingStack
{

 static StackMinimum stackMinimum1=new StackMinimum();
 static StackMinimum stackMinimum2=new StackMinimum();
 public static void enqueue(int a)
 {
  stackMinimum1.push(a);
 }

 public static int dequeue()
 {
  if(stackMinimum2.size()==0)
  {
   while(stackMinimum1.size()!=0)
    stackMinimum2.push(stackMinimum1.pop());
  }
  return stackMinimum2.pop();
 }

 public int getMinimum()
 {
  return Math.min(
   stackMinimum1.size()==0?
     Integer.MAX_VALUE:stackMinimum1.getMinimum(),
   stackMinimum2.size()==0?
     Integer.MAX_VALUE:stackMinimum2.getMinimum()
        );
 }
}
```

Find shortest path in a maze

Problem

Given a maze some of whose the cells are blocked. The left top cell is the entry point and right bottom cell is the exit point. Find the shortest path, if possible, from entry to exit through non blocked cells.
For example
Given maze

```
_|_|_|_|#|_|_|_|
_|_|#|_|#|_|_|#|#|
_|#|_|_|_|#|#|#|_|#|
_|#|_|_|_|#|_|_|#|_|
#|_|#|#|_|_|_|_|_|
#|_|_|_|_|_|#|_|_|
#|_|_|#|_|_|_|_|_|
#|_|_|_|_|_|_|#|_|
_|_|_|#|_|#|#|_|_|#|
#|#|#|#|_|#|_|_|_|
```

Shortest path
(0,1),(0,2),(0,3),(1,3),(2,3),(2,4),(3,4),(4,4),(4,5),(4,6),(5,6),(6,6),(6,7),(7,7),(8,7),(8,8),(9,8),(9,9)

Solution

For shortest path we do the breadth first traversal. We mark the entry point as 1, then all the accessible cells from 1 is marked as 2, then all the accessible cells from 2 that are not already marked are marked as 3. In this way any cell that is accessible from n marked node and not already marked are marked as n+1. In this way when the exit cell is marked. We find the shortest path.

Now to reconstruct the path we start from exit. Note its mark n and any adjacent cell that has marked with n-1 is the previous cell. Now we add previous cell in the path and search for n-2 in its adjacent cells. In this way when we reach the entry cell our path is constructed

Code

```
import java.util.LinkedList;
import java.util.List;

public class ShortestPathInMaze
{
 public static void main(String[] args)
 {
  boolean[][] maze = new boolean[10][10];
  makeRandomMaze(maze);
  printMaze(maze);
  List path = findShortestPath(maze);
  if (path == null)
  {
   System.out.println("No path possible");
   return;
  }
  for (Cell cell : path)
   System.out.print(cell + ",");
 }

 private static List findShortestPath(boolean[][] maze)
 {
  int[][] levelMatrix = new int[maze.length][maze[0].length];
  for (int i = 0; i < maze.length; ++i)
   for (int j = 0; j < maze[0].length; ++j)
   {
    levelMatrix[i][j] = maze[i][j] == true ? -1 : 0;
   }
  LinkedList < cell > queue = new LinkedList < cell >();
  Cell start = new Cell(0, 0);
  Cell end = new Cell(maze.length - 1, maze[0].length - 1);
  queue.add(start);
  levelMatrix[start.row][start.col] = 1;
  while (!queue.isEmpty())
  {
   Cell cell = queue.poll();
   if (cell == end)
    break;
   int level = levelMatrix[cell.row][cell.col];
   Cell[] nextCells = new Cell[4];
   nextCells[3] = new Cell(cell.row, cell.col - 1);
   nextCells[2] = new Cell(cell.row - 1, cell.col);
   nextCells[1] = new Cell(cell.row, cell.col + 1);
   nextCells[0] = new Cell(cell.row + 1, cell.col);

   for (Cell nextCell : nextCells)
   {
    if (nextCell.row < 0 || nextCell.col < 0)
     continue;
    if (nextCell.row == maze.length
      || nextCell.col == maze[0].length)
     continue;
    if (levelMatrix[nextCell.row][nextCell.col] == 0)
    {
     queue.add(nextCell);
     levelMatrix[nextCell.row][nextCell.col] = level + 1;
    }
```

```
   }
  }
  if (levelMatrix[end.row][end.col] == 0)
   return null;
  LinkedList < cell > path = new LinkedList < cell >();
  Cell cell = end;
  while (!cell.equals(start))
  {
   path.push(cell);
   int level = levelMatrix[cell.row][cell.col];
   Cell[] nextCells = new Cell[4];
   nextCells[0] = new Cell(cell.row + 1, cell.col);
   nextCells[1] = new Cell(cell.row, cell.col + 1);
   nextCells[2] = new Cell(cell.row - 1, cell.col);
   nextCells[3] = new Cell(cell.row, cell.col - 1);
   for (Cell nextCell : nextCells)
   {
    if (nextCell.row < 0 || nextCell.col < 0)
     continue;
    if (nextCell.row == maze.length
      || nextCell.col == maze[0].length)
     continue;
    if (levelMatrix[nextCell.row][nextCell.col] == level - 1)
    {
     cell = nextCell;
     break;
    }
   }
  }
  return path;
 }

 private static class Cell
 {
  public int row;
  public int col;

  public Cell(int row, int column)
  {
   this.row = row;
   this.col = column;
  }

  @Override
  public boolean equals(Object obj)
  {
   if (this == obj)
    return true;
   if ((obj == null) || (obj.getClass() != this.getClass()))
    return false;
   Cell cell = (Cell) obj;
   if (row == cell.row && col == cell.col)
    return true;
   return false;
  }

  @Override
  public String toString()
  {
   return "(" + row + "," + col + ")";
  }
 }

 private static void printMaze(boolean[][] maze)
 {
  for (int i = 0; i < maze.length; ++i)
  {
   for (int j = 0; j < maze[i].length; ++j)
```

```
   {
    if (maze[i][j])
     System.out.print("#|");
    else
     System.out.print("_|");
   }
   System.out.println();
  }
 }

 private static void makeRandomMaze(boolean[][] maze)
 {
  for (int i = 0; i < maze.length; ++i)
  {
   for (int j = 0; j < maze[0].length; ++j)
   {
    maze[i][j] = (int) (Math.random() * 3) == 1;
   }
  }
  maze[0][0] = false;
  maze[maze.length - 1][maze[0].length - 1] = false;

 }
}
```

All unique letter substring

Problem

Given a string find out the longest substring which has all unique letters, i.e. no letter is repeated.

Solution

We start by putting two pointers named startIndex and endIndex at the beginning of the string. As we move forward the end index new found letters are added in a set. If the letter is already found in the set, we start moving the start pointer and keep on removing the corresponding letters. When we found that the letter which caused the repetition is removed from the set, then we know that all the letters are unique in this substring. Every time we find such substrings we compare its length with a kept maximum. In this way we find the longest substring which does not have any repetition.

Code

```
import java.util.HashMap;
import java.util.HashSet;

public class AllUniqueSubstring
{
 public static void main(String[] args)
 {
  String input = "abacbdadbc";
  printAllUniqueSubstring(input);
 }

 public static void printAllUniqueSubstring(String input)
 {
  HashSet<Character> set = new HashSet<Character>();
  int startIndex = 0;
  int endIndex = 0;
  int maxLength = 0;
  int startMax = 0;
  int endMax = 0;

  while (endIndex < input.length())
  {
   char addChar = input.charAt(endIndex);
   if (!set.contains(addChar))
   {
    set.add(addChar);
    int currentLength = endIndex - startIndex + 1;
    if (currentLength > maxLength)
    {
     maxLength = currentLength;
     startMax = startIndex;
     endMax = endIndex;
    }
    endIndex++;
   } else
   {
```

```
    char removeChar = input.charAt(startIndex);
    set.remove(removeChar);
    startIndex++;
    if (removeChar == addChar)
    {
     int currentLength = endIndex - startIndex + 1;
     if (currentLength > maxLength)
     {
      maxLength = currentLength;
      startMax = startIndex;
      endMax = endIndex;
     }
     endIndex++;
    }
   }
  }
  System.out.println(startMax + "," + endMax);
  System.out.println(input.substring(startMax, endMax));
 }

}
```

Linked list remove duplicates

Problem

You have an infinite linked list which is not sorted and contains duplicate elements. You have to convert it to a linked list which does not contain duplicate elements.

Solution

While traversing through the input linked list we will maintain a hashset. If a number from input linked list is present in the hashset we will ignore it. otherwise we will insert it into hashset and inset into output linked list. The below program creates a similar representation of java LinkedHashSet. We can use that as well. Assuming a good hash distribution the space complexity is O(n) and time complexity is O(n).

Code

```
import java.util.HashSet;

public class LinkedListRemoveDuplicate
{
 public static void main(String[] args)
 {
  Chain input = new Chain();
  input.add(3).add(5).add(4).add(2).add(3).add(2).add(6).add(3).add(4);
  Chain output = removeDuplicate(input);
  for (Node node = output.head; node != null; node = node.next)
   System.out.print(node.value + "->");
  System.out.println();
 }

 private static Chain removeDuplicate(Chain input)
 {
  Chain output = new Chain();
  HashSet hashSet = new HashSet();
  for (Node node = input.head; node != null; node = node.next)
  {
   if (!hashSet.contains(node.value))
   {
    hashSet.add(node.value);
    output.add(node.value);
   }
  }
  return output;
```

```
 }

 public static class Node
 {
  int value;
  Node next;

  public Node(int value)
  {
   this.value = value;
  }
 }

 private static class Chain
 {
  Node head;
  Node last;

  public Chain add(int value)
  {
   Node node = new Node(value);
   if (head == null)
   {
    head = node;
    last = node;
   } else
   {
    last.next = node;
    last = node;
   }
   return this;
  }
 }
}
```

Find if two words are anagram or not

Problem

Given two words, find whether they are anagram or not. An anagram of a word is another word which can be obtained by rearranging the letters of the first word. Without any addition or deletion of letters.

Solution

We will use the property of anagrams that the count of different letters in anagrams are equal. So we will take one word and keep its letter's count in a hash map. Now for second word we will keep on decreasing the count form the map. When the count is zero, we will remove the key from the map. In this way if two words are anagram then nothing should be left in the map at the end of this operation. So we return true if the map has no keys left. If during removal we find that that some letter is not found for removal then also it is not an anagram.

Code

```
import java.util.HashMap;
import java.util.Map;

public class OneWordPermutationOfOther
{
 public static void main(String[] args)
 {
  String a = "listen";
  String b = "enlist";
```

```
  System.out.println(isPermutation(a, b));
 }

 private static boolean isPermutation(String a, String b)
 {
  Map < Character, Integer> charMap = new HashMap < Character, Integer>();
  for (char ch : a.toCharArray())
  {
   if (charMap.containsKey(ch))
    charMap.put(ch, charMap.get(ch) + 1);
   else
    charMap.put(ch, 1);
  }
  for (char ch : b.toCharArray())
  {
   if (!charMap.containsKey(ch))
    return false;
   if (charMap.get(ch) == 1)
    charMap.remove(ch);
   else
    charMap.put(ch, charMap.get(ch) - 1);
  }
  if (charMap.keySet().size() == 0)
   return true;
  return false;
 }

}
```

Longest subarray with equal number of ones and zeros

Problem

You are given an array of 1's and 0's only. Find the longest subarray which contains equal number of 1's and 0's.

Solution

We will keep a running sum of "no of ones minus no of zeros" for each index of the array. For any two indices, if the running sum is equal, that means there are equal number of ones and zeros between these two indices. We will store the running sum in an array such that it acts like a hash map where key is the running sum and value is the leftmost index of that running sum to appear. The running sum can vary from -n to +n. So we need an array of length 2*n+1. So for any index we can check whether this sum has occurred before and if yes what is the left most index for it. We can do this in O(1) time by maintaining the array. So at any index we can find the longest equal subarray till that point. We will compare it with a maximum value and update the maximum accordingly. So the overall complexity for the process is O(n)

Code

```
public class MaxSubarrayEqualOnesZeros
{
 public static void main(String[] args)
 {
  int[] arr =
  { 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0,
     1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0 };
  printMaxSubarray(arr);
 }

 private static void printMaxSubarray(int[] arr)
 {
  Integer[] diffMap = new Integer[arr.length * 2 + 1];
  diffMap[arr.length] = -1;
```

```
  int sum = 0;
  int maxLength = 0;
  int maxStart = -1;
  int maxEnd = -1;
  for (int i = 0; i < arr.length; ++i)
  {
   if (arr[i] == 0)
    sum -= 1;
   else
    sum += 1;
   Integer prevIndex = diffMap[arr.length + sum];
   if (prevIndex == null)
    diffMap[arr.length + sum] = i;
   else
   {
    if (i - prevIndex > maxLength)
    {
     maxLength = i - prevIndex;
     maxStart = prevIndex + 1;
     maxEnd = i;
    }
   }
  }
  System.out.println("indices (" + maxStart + "," + maxEnd + ")");
  System.out.println("length=" + maxLength);
 }
}
```

Find the Pythagorean triplets in an array

Problem

You are given an integer array. Find all the Pythagorean triplets in this array. Pythagorean triplets are 3 numbers a, b, c such that a^2+b^2=c^2.

Solution

First we store all the squares of the numbers in a hash set. Then in O(n^2) we try with all the combinations (a,b) and search a^2+b^2 in the hash set. If it exists then we have found a pythagorean triplet. The complexity is O(n^2) as compared to brute force of O(n^3).

Code

```
package com.dsalgo;

import java.util.HashSet;

public class PythagoreanTriplet {

 public static void main(String[] args) {
  int[] arr = { 2, 3, 4, 6, 7, 12, 13, 15, 5, 17, 14, 22 };
  findPythagoreanTriplets(arr);

 }

 private static void findPythagoreanTriplets(int[] arr) {
  HashSet squares = new HashSet();
  for (int num : arr)
   squares.add((long) num * num);
  for (int i = 0; i < arr.length - 1; ++i)
   for (int j = i + 1; j < arr.length; ++j) {
```

```
    long square = (long) arr[i] * arr[i] + (long) arr[j] * arr[j];
    if (squares.contains(square)) {
     System.out.println(arr[i] + "," + arr[j] + ","
       + (long) Math.sqrt(square));
    }
   }
 }

}
```

Linked list with random pointer

Problem

A linked list is given where every node of the linked list has one next pointer and one additional pointer called random, which points to any arbitrary node in the linked list. Write a method to deep copy this linked list

Solution

First the linked list is normally deep copied to a new linked list. While creating every node of the destination linked list, we create a map where key is the source node and value is the corresponding destination node. This map is later used to populate the random pointer. We again traverse the source linked list. For each node we take the node pointed by its random pointer and search it in the map, as the value is the corresponding node in destination linked list, we populate the random pointer accordingly.

Code

```
import java.util.HashMap;
import java.util.Map;


public class LinkedListRandomPointer
{

 public static void main(String[] args)
 {
  NodeRandom a=new NodeRandom(4);
  NodeRandom b=new NodeRandom(5);
  NodeRandom c=new NodeRandom(3);
  NodeRandom d=new NodeRandom(6);
  NodeRandom e=new NodeRandom(7);
  NodeRandom f=new NodeRandom(2);
  NodeRandom g=new NodeRandom(9);
  NodeRandom h=new NodeRandom(3);
  NodeRandom i=new NodeRandom(7);
  a.next=b;
  b.next=c;
  c.next=d;
  d.next=e;
  e.next=f;
  f.next=g;
  g.next=h;
  h.next=i;
  a.random=g;
  b.random=a;
  c.random=e;
  d.random=f;
  e.random=e;
  f.random=null;
  g.random=b;
  h.random=i;
  i.random=f;
  printLinkedList(a);
  printLinkedList(deepCopy(a));

 }
 private static void printLinkedList(NodeRandom head)
 {
  while(head!=null)
  {
   System.out.print(head);
   head=head.next;
  }
  System.out.println();
 }
 public static NodeRandom deepCopy(NodeRandom sourceHead)
 {
  Map<NodeRandom,NodeRandom> map=
    new HashMap<NodeRandom, NodeRandom>();
  NodeRandom sourcePtr=sourceHead;
  NodeRandom destHead=null;
```

```
  NodeRandom destPtr=null;
  while(sourcePtr!=null)
  {
   if(destHead==null)
   {
    destHead=new NodeRandom(sourcePtr.value);
    destPtr=destHead;
   }
   else
   {
    destPtr.next=new NodeRandom(sourcePtr.value);
    destPtr=destPtr.next;
   }
   map.put(sourcePtr, destPtr);
   sourcePtr=sourcePtr.next;
  }
  sourcePtr=sourceHead;
  destPtr=destHead;
  while(sourcePtr!=null)
  {
   destPtr.random=map.get(sourcePtr.random);
   sourcePtr=sourcePtr.next;
   destPtr=destPtr.next;
  }
  return destHead;
 }

 static class NodeRandom
 {
  public NodeRandom next;
  public NodeRandom random;
  public int value;
  public NodeRandom(int value)
  {
   this.value=value;
  }
  public String toString()
  {
   if(random==null)
    return "{"+hashCode()+"("+value+") r->null}";
   else
    return "{"+hashCode()+"("+value+") r->"+
      random.hashCode()+"}";
  }
 }
}
```

Same average subset

Problem

Given an integer array find all subsets having equal average.

Solution

We will use a structure to store the count and sum of a subset. So that if we add any element to an existing array we can find the average in O(1) time. After processing till ith index we memoize all the subsets possible with the elements from 0 to i. Now for i+1 th number we add this number to all previous subsets and start a subset with the number itself. As we maintain the sum and count for all the subsets, we can calculate the average at the same complexity as we add the numbers to the old subsets. Then we create a hashmap where key is the average and value is a list of subsets. We add all the subsets in this hash map. Then we iterate over all the keys of the hash map and wherever the value contains a list having more than one element we print those subsets corresponding to that average. This has a complexity of O(n*2^n). Because there are total 2^n possible subsets from n numbers.

Code

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
```

```
public class SameAverageSubset
{
 public static void main(String[] args)
 {
  int[] input =
  { 4, 1, 7, 8, 3, 5 };
  ArrayList table = new ArrayList();
  HashMap < Double, List> hashMap = new HashMap < Double, List>();
  for (int i = 0; i < input.length; ++i)
  {
   int num = input[i];
   ArrayList tempTable = new ArrayList();
   for (SumCountPair j : table)
   {
    SumCountPair temp = j.clone();
    temp.add(num);
    tempTable.add(temp);
   }
   table.addAll(tempTable);
   SumCountPair sumCountPair = new SumCountPair();
   sumCountPair.add(num);
   table.add(sumCountPair);
  }
  for (SumCountPair j : table)
  {
   List temp = hashMap.get(j.average());
   if (temp != null)
   {
    temp.add(j);
   } else
   {
    temp = new ArrayList();
    temp.add(j);
   }
   hashMap.put(j.average(), temp);
  }
  for (Double average : hashMap.keySet())
  {
   List list = hashMap.get(average);
   if (list.size() == 1)
    continue;
   System.out.print(average + " ");
   for (SumCountPair sumCountPair : list)
    System.out.print(sumCountPair + ", ");
   System.out.println();
  }

 }

}

class SumCountPair
{
 int sum;
 int count;
 ArrayList array;

 public SumCountPair()
 {
  this(0, 0);
 }

 public SumCountPair(int sum, int count)
 {
  this(sum, count, new ArrayList());
 }

 public SumCountPair(int sum, int count, ArrayList array)
```

```
 {
  this.sum = sum;
  this.count = count;
  this.array = (ArrayList) array.clone();
 }

 @Override
 public SumCountPair clone()
 {
  return new SumCountPair(sum, count, array);
 }

 public void add(int num)
 {
  sum += num;
  count++;
  array.add(num);
 }

 public double average()
 {
  return (double) sum / count;
 }

 public String toString()
 {
  return array.toString();
 }
}
```

Permutation of a string as substring of another

Problem

Given a string A and another string B, Find whether any permutation of B exists as a substring of A.
For example,
if A = "encyclopedia"
if B="dep" then return true as ped is a permutation of dep and ped is a substring of A
if B="ycc" return true as cyc is a substring
if B="yyc" return false.

Solution

We need to find whether a permutation of B exists as a substring of A. So there would be a substring of length of B in the string A which has exact same letter count as B. We will calculate the letter count of B in a hashmap. We will maintain 3 variables, existing letter positive count, existing letter negative count and non-existing letter. We will first initialize the hash map and the existing letter positive count with the letters of B. Now we will iterate over A with a window of length of B. As we come across new letters, we will decrease its count from the hash map. As letters are excluded from the window we will again add it to the hashmap. If at any point the hashmap contains zero for all letters, then we can say that in the given window we have found a substring which has exactly same count of letters as the word B. So we return true. If the length of B is k then we can search the hashmap and find out whether all letter count is zero or not in O(k) time. But by maintaining three extra variable we can do it in O(1) time. Whenever a letter count goes down by 1 but still stays positive we decrease the existing count positive. Whenever it becomes negative we increase the existing negative count. Similarly when letter count becomes 0 from -1 we again decrease the existing negative count. When some letter that is not present in the hashmap comes in the considered window we decrease non-existing count. And when it leaves the window we increase it. So when non-existing count becomes zero that means the considered window has only letters of B. Now if existing positive count and existing negative count becomes zero, that means all the letter count are zero. Why do we need positive and negative count? This is because one letter count can become -1 and one letter count becomes +1, then sum will be zero. although the substring is not having exact letter count for all letters.

Code

```
import java.util.HashMap;

public class PermutationSubstring
{
 public static void main(String[] args)
 {
  String a = "encyclopedia";
  String b = "dope";
  System.out.println(isPermutationSubstring(a, b));
 }

 private static boolean isPermutationSubstring(String a, String b)
 {
  if (a.length() < b.length())
   return false;
  int sameLetterPositive = b.length();
  int sameLetterNegative = 0;
  int otherLetter = 0;
  HashMap hashMap = new HashMap();
  for (char ch : b.toCharArray())
  {
   if (hashMap.containsKey(ch))
   {
    hashMap.put(ch, hashMap.get(ch) + 1);
   } else
   {
    hashMap.put(ch, 1);
   }
  }
  for (int i = 0; i < b.length(); ++i)
  {
   char ch = a.charAt(i);
   if (hashMap.containsKey(ch))
   {
    int count = hashMap.get(ch);
    hashMap.put(ch, count - 1);
    if (count == 0)
     sameLetterNegative++;
    else
     sameLetterPositive--;
   } else
   {
    otherLetter--;
   }
  }
  if (sameLetterPositive == 0 && sameLetterNegative == 0
    && otherLetter == 0)
   return true;
  for (int i = b.length(); i < a.length(); ++i)
  {
   char inclusion = a.charAt(i);
   char exclusion = a.charAt(i - b.length());
   if (hashMap.containsKey(exclusion))
   {
    int count = hashMap.get(exclusion);
    hashMap.put(exclusion, count + 1);
    if (count == -1)
     sameLetterNegative--;
    else
     sameLetterPositive++;
   } else
   {
    otherLetter++;
   }
   if (hashMap.containsKey(inclusion))
   {
    int count = hashMap.get(inclusion);
    hashMap.put(inclusion, count - 1);
```

```
    if (count == 0)
     sameLetterNegative++;
    else
     sameLetterPositive--;
   } else
   {
    otherLetter--;
   }
   if (sameLetterPositive == 0 && sameLetterNegative == 0
     && otherLetter == 0)
    return true;
  }
  return false;
 }
}
```

Balance the balance

Problem

There are some weighing scales which has left and right plates. The weighing scales are very compact and plates are pretty big. As a result of this, one weighing scale can be put into another weighing scale's plate. At the same time some weights can be put onto the plates along with the weighing scale. The empty weighing scale has a fixed predefined weight. Only one weighing scale can be accommodated in one plate. However any number of weights can be put into the plate, even after putting one weighing scale. One such arrangement is given as an input to your program. There are enough supply of weights and they can be of fractional value as well. The program should add weights to the plates to balance the weighing scales. The ultimate result would be that all the weighing scales are balanced.

Solution

We can balance the weighing scale which has no more weighing scales in its left or right plate. When these are balanced, we can try to balance the weighing scales which has no unbalanced weighing scales in their plates. So if we represent the weighing scale like a binary tree, the ones with only weights will be it's leaf nodes and the weighing scale which is not contained in any other's plate will be the root node. Now if we traverse the tree in a manner that root comes after it's children and we recursively traverse the tree we can find our solution. So the problem is solved by representing the arrangement of the weighing scales by binary tree and then by doing a post order traversal to balance the balances.

Code

```
public class BalanceWeight
{
 final static int weightOfBalanceBeam = 10;

 public static void main(String[] args)
 {
  Balance a = new Balance(1, 4, 6);
  Balance b = new Balance(2, 3, 8);
  Balance c = new Balance(3, 5, 9);
  Balance d = new Balance(4, 3, 2);

  a.left = b;
  a.right = c;
  b.left = d;

  doBalance(a);
 }

 public static double doBalance(Balance balance)
 {
  if (balance == null)
   return 0;
  if (balance.left == null && balance.right == null)
  {
   return printBalance(balance.id, balance.leftWeight,
    balance.rightWeight);
  }
  else
  {
   return printBalance(balance.id, balance.leftWeight
     + doBalance(balance.left), balance.rightWeight
     + doBalance(balance.right));
  }
 }

 public static double printBalance(int id, double leftWeight,
   double rightWeight)
 {
  if (leftWeight > rightWeight)
  {
   System.out.println("Add " + (leftWeight - rightWeight) +
    " to the right of Balance#" + id);
   return leftWeight * 2 + weightOfBalanceBeam;
  }
  else if (leftWeight < rightWeight)
```

```
  {
   System.out.println("Add " + (rightWeight - leftWeight) +
    " to the left of Balance#" + id);
   return rightWeight * 2 + 10;
  }
  else
  {
   return rightWeight * 2 + 10;
  }
 }

}
class Balance
{
 public int id;
 public Balance left;
 public Balance right;
 public double leftWeight;
 public double rightWeight;

 public Balance(int id, double leftWeight, double rightWeight)
 {
  this.id = id;
  this.leftWeight = leftWeight;
  this.rightWeight = rightWeight;
 }
}
```

Lowest common ancestor

Problem

A binary tree is given and two nodes of this tree is given, we have to find out the algorithm for lowest common ancestor of the two given nodes. Common ancestor are the nodes which can be reached from two nodes by following the parent links. Lowest common ancestor is the common ancestor which has the highest depth.

Solution

A node X has left subtree and right subtree. One or both of them can be null. If the two given nodes A and B are not present in any of the subtrees, then X can never be reached from A and B following the parent link. On the other hand if A and B are both on the right subtree or in the left subtree, then the node X is common ancestor, but not the lowest one. Because if A and B are both present in the right subtree of X, then at least root of the right subtree of X is a better candidate for being lowest common ancestor.
So lowest common ancestor will be a node which has one given node on its left and the other one on its right. Or the node itself is one of the given node and another one is in its subtree. In this solution we have a recursive function which tries to find the count of found nodes in a node's left and right subtree. When the numbers are 1 on left and 1 on right then the given node is the answer. When the count is 1 in either side and the node itself is one of the given node then also the node is the lowest common ancestor, else it recursively calls the function to its left and right children and return the sum of them.

Code

```
package problems;

public class LowestCommonAncestor
{

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

  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(8);
  Node g = new Node(6);
  Node h = new Node(7);
  a.left = b;
  a.right = c;
  b.left = d;
  c.left = e;
  c.right = f;
  f.left = g;
  f.right = h;
  LCA(a, c, h);
  System.out.println(LCANode.value);

 }

 static Node LCANode = null;

 public static int LCA(Node currentNode, Node n1, Node n2)
 {
  if (currentNode == null)
   return 0;
```

```
  int currentValue = 0;
  if (currentNode == n1 || currentNode == n2)
   currentValue = 1;
  int leftValue = LCA(currentNode.left, n1, n2);
  int rightValue = LCA(currentNode.right, n1, n2);
  if ((currentValue == 1 && leftValue == 1)
     ||(currentValue == 1 && rightValue == 1)
     ||(leftValue == 1 && rightValue == 1))
  {
   LCANode = currentNode;
   return 2;
  }
  return currentValue + leftValue + rightValue;

 }

}
```

Sum of child nodes

Problem

In a binary tree change each node's value(except leaf node) as the sum of left and right subtree's value.

```
           Constructed binary tree is:
                  1
                /     \
               2       3
             /  \       \
            4   5       8
                      /   \
                     6     7


         After sum stored is in each node, binary tree is:
                 35
               /    \
              9     21
            /  \      \
           4   5      13
                    /   \
                   6     7
```

Solution

In a recursive function we return the sum of the subtree rooted at a particular node and change the value of the node also. If the node is null return 0, otherwise it calls it recursively to its left and right children. It returns this node's value plus the results returned by calling the function on its left and right children. And sets the value of the current node to the sum of the left and right side.

Code

```
/*
 * In a binary tree change each node's value(except leaf node)
 * as the sum of left and right subtree's value.
 *
   Constructed binary tree is:
           1
         /     \
        2       3
      /  \       \
     4   5       8
               /   \
              6     7


 After sum stored is in each node, binary tree is:
          35
        /    \
       9     21
     /  \       \
    4   5       13
              /   \
             6     7

 */
public class BinaryTreeSumChildNodes
{

 /**
  * @param args
```

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

  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(8);
  Node g = new Node(6);
  Node h = new Node(7);
  a.left = b;
  a.right = c;
  b.left = d;
  b.right = e;
  c.right = f;
  f.left = g;
  f.right = h;
  printNice(a);
  System.out.println();
  makeSum(a);
  printNice(a);
 }

 public static int makeSum(Node root)
 {
  if (root == null)
   return 0;
  int temp = root.value;
  int sum = makeSum(root.left) + makeSum(root.right);
  if (root.left != null || root.right != null)
   root.value = sum;
  return temp + sum;
 }

 public static void printNice(Node root)
 {
  if (root == null)
   return;
  else
  {
   System.out.print(root.value);
   if (root.left != null)
   {
    System.out.print("L->[");
    printNice(root.left);
    System.out.print("]");
   }
   if (root.right != null)
   {
    System.out.print("R->[");
    printNice(root.right);
    System.out.print("]");
   }
  }
 }
}
```

Lowest common ancestor without root

Problem

There is a binary tree where each node is having a parent pointer. We need to have an algorithm to find out the lowest common ancestor where two nodes are given, but no root node is given.

Solution

We take one node and follow its parent link till we face null. We keep on putting each node in a hashset. Next we take another node and follow its parent link and keep on checking each node in the hashset. when it finds a match in hashset, that very node is the lowest common ancestor.

Code

```
import java.util.HashSet;


public class LeastCommonAncestorWithoutRoot
{
 public static void main(String[] args)
 {
  NodeWithParent a=new NodeWithParent(5);
  NodeWithParent b=new NodeWithParent(6);
  NodeWithParent c=new NodeWithParent(7);
```

```
  NodeWithParent d=new NodeWithParent(8);
  NodeWithParent e=new NodeWithParent(9);
  a.left=b;
  b.parent=a;
  b.left=c;
  c.parent=b;
  b.right=d;
  d.parent=b;
  d.right=e;
  e.parent=d;
  System.out.println("LCA: "+getLCA(c, e).value);

 }

 public static NodeWithParent getLCA(NodeWithParent a,
  NodeWithParent b)
 {
  HashSet<NodeWithParent> table=
    new HashSet<NodeWithParent>();
  while(a!=null)
  {
   table.add(a);
   a=a.parent;
  }
  while(b!=null)
  {
   if(table.contains(b))
    return b;
   b=b.parent;
  }
  return null;
 }

}

class NodeWithParent
{
 public NodeWithParent parent;
 public int value;
 public NodeWithParent left;
 public NodeWithParent right;

 public NodeWithParent(int value)
 {
  this.value=value;
 }
}
```

Binary tree zigzag print

Problem

Given a binary tree print it in level order zigzag form

Solution

A level order traversal of a tree is done using queue. Root is inserted into the queue. Then the iteration is done till the queue is empty. While a node is dequeued, its left and right children are added to the queue. In this way a normal level order print is possible. For zigzag, we keep a marker node. After root, the marker is inserted into the queue. So whenever marker is dequeued, that means a level has ended, so its children are already in the queue. So at this point the next level is also inserted. So we put the marker back into the queue if the queue is not already empty. For a reverse print of the level we use a stack to store the nodes, till the level is done.

Code

```
import java.util.LinkedList;
import java.util.LinkedList;
import java.util.Stack;

public class BinaryTreeZigzag {
 public static void main(String[] args) {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(6);
  Node g = new Node(7);
  Node h = new Node(8);
  Node i = new Node(9);

  a.left = b;
  a.right = c;
  b.right = d;
  c.left = e;
```

```
  c.right = f;
  d.left = g;
  d.right = h;
  g.right = i;

  printZigzag(a);

 }

 public static void printZigzag(Node root) {
  LinkedList queue = new LinkedList();
  Stack stack = new Stack();
  if (root == null)
   return;
  Node marker = new Node(-1);
  boolean printOrder = true;
  queue.add(root);
  queue.add(marker);
  while (!queue.isEmpty()) {
   Node node = queue.poll();
   if (node == marker) {
    if (!printOrder) {
     while (stack.size() > 0) {
      Node printNode = stack.pop();
      System.out.println(printNode.value);

     }

    }
    printOrder = !printOrder;
    if (!queue.isEmpty())
     queue.add(marker);
    continue;
   }
   if (printOrder) {
    System.out.println(node.value);
   } else {
    stack.push(node);
   }
   if (node.left != null)
    queue.add(node.left);
   if (node.right != null)
    queue.add(node.right);

  }

 }

 static class Node {
  public Node left;
  public Node right;
  public int value;

  public Node(int value) {
   this.value = value;
  }
 }

}
```

Print a binary tree from bottom to top

Problem

Print a binary tree from bottom to top in level order. For example,

```
Given binary tree is:
        1
      /   \
     2     3
   /  \     \
  4   5     6
           /   \
          7     8
```

```
The output will be 7,8,4,5,6,2,3,1
```

Solution

We will do normal level order traversal using a queue, while doing that we will keep on adding the nodes to a stack. So that it reverses during print. As we will print it from left to right, while putting it into the queue for level order traversal, we will put right node first and then the left node, so that after reversing from the stack it will become left, right.

Code

```
import java.util.List;
import java.util.Stack;

public class PrintBinartyTreeBottomToTop
{
 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(6);
  Node g = new Node(7);
  Node h = new Node(8);
  a.left = b;
  a.right = c;
  b.left = d;
  c.left = e;
  c.right = f;
  f.left = g;
  f.right = h;

  printLevelFromBottomToTop(a);
 }


 private static void printLevelFromBottomToTop(Node root)
 {
  LinkedList < node > queue=new LinkedList < node >();
  Stack < node > stack=new Stack < node >();
  queue.add(root);
  while(!queue.isEmpty())
  {
   Node node=queue.poll();
   stack.push(node);
   if(node.right!=null)
   {
    queue.add(node.right);
   }
   if(node.left!=null)
   {
    queue.add(node.left);
   }
  }
  while(!stack.isEmpty())
  {
   System.out.print(stack.pop().value+", ");
  }
 }


 static class Node
 {
  Node left;
  Node right;
  int value;
```

```
  public Node(int value)
  {
   this.value = value;
  }
 }
}
```

Print binary tree bottom to top level wise

Problem

Print a binary tree from bottom to top in level order each level in a separate row. For example,

```
            Given binary tree is:
                    1
                  /   \
                 2     3
               /  \     \
              4   5     6
                       /   \
                      7     8


      The output will be
7,8,
4,5,6,
2,3,
1
```

Solution

Like the previous program of printing binary tree from bottom to top, in this one we will keep a level marker in the stack to break the lines after a level

Code

```
import java.util.LinkedList;
import java.util.Stack;

public class PrintBinartyTreeBottomToTop
{
 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(6);
  Node g = new Node(7);
  Node h = new Node(8);
  a.left = b;
  a.right = c;
  b.left = d;
  c.left = e;
  c.right = f;
  f.left = g;
  f.right = h;

  printLevelFromBottomToTop(a);
 }

 private static void printLevelFromBottomToTop(Node root)
 {
  Node levelMarker = new Node(-1);
```

```
  LinkedList < node > queue = new LinkedList < node >();
  Stack < node > stack = new Stack < node >();
  queue.add(root);
  queue.add(levelMarker);
  while (!queue.isEmpty())
  {
   Node node = queue.poll();
   stack.push(node);
   if (node == levelMarker)
   {
    if (queue.isEmpty())
     break;
    queue.add(levelMarker);
    continue;
   }

   if (node.right != null)
   {
    queue.add(node.right);
   }
   if (node.left != null)
   {
    queue.add(node.left);
   }
  }
  while (!stack.isEmpty())
  {
   Node node = stack.pop();
   if (node == levelMarker)
   {
    System.out.println();
    continue;
   }
   System.out.print(node.value + "\t ");
  }
 }

 static class Node
 {
  Node left;
  Node right;
  int value;

  public Node(int value)
  {
   this.value = value;
  }
 }
}
```

Print nodes of a given level

Problem

Given a binary tree and a level, print all the nodes of the corresponding level.

Solution

We call a recursive function on the root of the binary tree by passing the given level. We do a preorder traversal of the binary tree, every time we call the recursive function on any node's children, we decrease the level value by one. When level value is zero, that means we reached at the correct level. Then we print the node's value and stop proceeding to further depths.

Code

```
public class PrintNodeOfSameLevel
```

```
{
 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(6);
  Node g = new Node(7);
  Node h = new Node(8);
  a.left = b;
  a.right = c;
  b.left = d;
  c.left = e;
  c.right = f;
  f.left = g;
  f.right = h;

  printLevelofDepth(a, 5);
 }

 private static void printLevelofDepth(Node a, int depth)
 {
  if (a == null)
   return;
  if (depth == 1)
  {
   System.out.println(a.value);
   return;
  }
  printLevelofDepth(a.left, depth - 1);
  printLevelofDepth(a.right, depth - 1);
 }

 static class Node
 {
  Node left;
  Node right;
  int value;

  public Node(int value)
  {
   this.value = value;
  }
 }
}
```

Linked list with inorder successor

Problem

Create a linked list with the nodes of a binary tree in an inorder succession.

Solution

We traversed the tree recursively in inorder traversal. While doing so, we took two other variables, named linker and head. When the first node is visited in an inorder traversal the head node and linker are initialized. Then for next inorder successors, head is never touched but linker is used to link the next elements.

Code

```
public class InorderSuccessor
{

 public static void main(String[] args)
 {
  NextNode a=new NextNode(1);
  NextNode b=new NextNode(2);
  NextNode c=new NextNode(3);
  NextNode d=new NextNode(4);
```

```
  NextNode e=new NextNode(5);
  NextNode f=new NextNode(6);
  NextNode g=new NextNode(7);
  NextNode h=new NextNode(8);
  NextNode i=new NextNode(9);
  a.left=b;
  a.right=c;
  b.right=d;
  c.left=e;
  c.right=f;
  d.left=g;
  d.right=h;
  g.right=i;
  NextNode head=linkInorderSuccessor(a);
  while(head!=null)
  {
   System.out.println(head.value);
   head=head.next;
  }

 }

 public static NextNode linkInorderSuccessor(NextNode root)
 {
  NodeContainer linker=new NodeContainer();
  NodeContainer head=new NodeContainer();
  linkInorderSuccessor(root, linker, head);
  return head.node;
 }

 private static void linkInorderSuccessor(NextNode root, NodeContainer linker, NodeContainer head)
 {
  if(root==null)
   return;
  linkInorderSuccessor(root.left,linker,head);
  if(head.node==null&&root!=null)
   head.node=root;
  if(linker.node!=null)
   linker.node.next=root;
  linker.node=root;
  linkInorderSuccessor(root.right, linker, head);
 }

}

class NextNode
{
 public NextNode left;
 public NextNode right;
 public NextNode next;
 public int value;
 public NextNode(int value)
 {
  this.value=value;
 }
}

class NodeContainer
{
 public NextNode node;
}
```

Linked list with preorder successor

Problem

Create a linked list with the nodes of a binary tree in an preorder succession.

Solution

Like the previous program, a linker is used to link the next nodes in the linked list while doing a preorder traversal. The first found node is stored in the head pointer.

Code

```
public class PreorderSuccessor
{

 public static void main(String[] args)
 {
  NextNode a=new NextNode(1);
  NextNode b=new NextNode(2);
  NextNode c=new NextNode(3);
```

```
  NextNode d=new NextNode(4);
  NextNode e=new NextNode(5);
  NextNode f=new NextNode(6);
  NextNode g=new NextNode(7);
  NextNode h=new NextNode(8);
  NextNode i=new NextNode(9);
  a.left=b;
  a.right=c;
  b.right=d;
  c.left=e;
  c.right=f;
  d.left=g;
  d.right=h;
  g.right=i;
  NextNode head=linkPreorderSuccessor(a);
  while(head!=null)
  {
   System.out.println(head.value);
   head=head.next;
  }

 }

 public static NextNode linkPreorderSuccessor(NextNode root)
 {
  NodeContainer linker=new NodeContainer();
  NodeContainer head=new NodeContainer();
  linkPreorderSuccessor(root, linker, head);
  return head.node;
 }

 private static void linkPreorderSuccessor(NextNode root,
    NodeContainer linker, NodeContainer head)
 {
  if(root==null)
   return;
  if(head.node==null&&root!=null)
   head.node=root;
  if(linker.node!=null)
   linker.node.next=root;
  linker.node=root;
  linkPreorderSuccessor(root.left,linker,head);
  linkPreorderSuccessor(root.right, linker, head);
 }

}

class NextNode
{
 public NextNode left;
 public NextNode right;
 public NextNode next;
 public int value;
 public NextNode(int value)
 {
  this.value=value;
 }
}

class NodeContainer
{
 public NextNode node;
}
```

Linked list with postorder successor

Problem

Create a linked list with the nodes of a binary tree in a postorder succession.

Solution

Like the inorder successor program, a linker is used to link the linked list during post order traversal. The first found node is set to the head.

Code

```
public class PostorderSuccessor
{

 public static void main(String[] args)
 {
  NextNode a=new NextNode(1);
```

```
  NextNode b=new NextNode(2);
  NextNode c=new NextNode(3);
  NextNode d=new NextNode(4);
  NextNode e=new NextNode(5);
  NextNode f=new NextNode(6);
  NextNode g=new NextNode(7);
  NextNode h=new NextNode(8);
  NextNode i=new NextNode(9);
  a.left=b;
  a.right=c;
  b.right=d;
  c.left=e;
  c.right=f;
  d.left=g;
  d.right=h;
  g.right=i;
  NextNode head=linkPostorderSuccessor(a);
  while(head!=null)
  {
   System.out.println(head.value);
   head=head.next;
  }

 }

 public static NextNode linkPostorderSuccessor(
  NextNode root)
 {
  NodeContainer linker=new NodeContainer();
  NodeContainer head=new NodeContainer();
  linkPostorderSuccessor(root, linker, head);
  return head.node;
 }

 private static void linkPostorderSuccessor(
  NextNode root, NodeContainer linker, NodeContainer head)
 {
  if(root==null)
   return;
  linkPostorderSuccessor(root.left,linker,head);
  linkPostorderSuccessor(root.right, linker, head);
  if(head.node==null&&root!=null)
   head.node=root;
  if(linker.node!=null)
   linker.node.next=root;
  linker.node=root;

 }

}

class NextNode
{
 public NextNode left;
 public NextNode right;
 public NextNode next;
 public int value;
 public NextNode(int value)
 {
  this.value=value;
 }
}

class NodeContainer
{
 public NextNode node;
}
```

Binary tree to linked list

Problem

Given a binary tree create individual linked lists from each level of the binary tree.

Solution

We will do the level order traversal of the binary tree with a marker node to tell us when a level ends. While doing so we will keep the node values added to the linked lists. After each level we will add the linked list to the result and crate a new one for the next level.

Code

```
import java.util.ArrayList;
```

```
import java.util.LinkedList;
import java.util.List;


public class BinaryTreeToLinkedList
{

 public static void main(String[] args)
 {
  BinaryTreeNode a=new BinaryTreeNode(1);
  BinaryTreeNode b=new BinaryTreeNode(2);
  BinaryTreeNode c=new BinaryTreeNode(3);
  BinaryTreeNode d=new BinaryTreeNode(4);
  BinaryTreeNode e=new BinaryTreeNode(5);
  BinaryTreeNode f=new BinaryTreeNode(6);
  BinaryTreeNode g=new BinaryTreeNode(7);
  BinaryTreeNode h=new BinaryTreeNode(8);
  BinaryTreeNode i=new BinaryTreeNode(9);
  a.left=b;
  a.right=c;
  b.right=d;
  c.left=e;
  c.right=f;
  d.left=g;
  d.right=h;
  g.right=i;

  List<LinkedNode> result=getLinkedListFromEachLevel(a);
  for(LinkedNode head:result)
   printLinkedList(head);

 }

 static List<LinkedNode> getLinkedListFromEachLevel(
  BinaryTreeNode root)
 {
  List<LinkedNode> result=
    new ArrayList<LinkedNode>();
  if(root==null)
   return result;
  LinkedList<BinaryTreeNode> queue=
    new LinkedList<BinaryTreeNode>();
  queue.add(root);
  BinaryTreeNode marker=new BinaryTreeNode(-1);
  queue.add(marker);
  LinkedNode head=null;
  LinkedNode prev=null;
  while(!queue.isEmpty())
  {
   BinaryTreeNode btNode=queue.poll();
   if(btNode==marker)
   {
    result.add(head);
    head=null;

    if(!queue.isEmpty())
    {
     queue.add(marker);
    }
    continue;
   }
   if(head==null)
   {
    head=new LinkedNode(btNode.value);
    prev=head;
   }
   else
   {
    prev.next=new LinkedNode(btNode.value);
    prev=prev.next;
   }
   if(btNode.left!=null)
    queue.add(btNode.left);
   if(btNode.right!=null)
    queue.add(btNode.right);

  }
  return result;
 }
 static class BinaryTreeNode
 {
```

```
 public BinaryTreeNode left;
 public BinaryTreeNode right;
 public int value;
 public BinaryTreeNode(int value)
 {
  this.value=value;
 }
}

static class LinkedNode
{
 public LinkedNode next;
 public int value;
 public LinkedNode(int value)
 {
  this.value=value;
 }
}

static void printLinkedList(LinkedNode head)
{
 while(head!=null)
 {
  System.out.print(head.value);
  System.out.print("->");
  head=head.next;
 }
 System.out.println();
}

}
```

Is the binary tree BST

Problem

Find whether a given binary tree is a binary search tree or not.

Solution

We create a recursive function which takes max and min limit for its value. First the given root is tried within the range Integer.MAX and Integer.MIN. If they are fine for a BST, then this recursive function is tried for left and right subtree. When we try for left child the upper limit is the value of current node and whatever min is passed from parent. For right side min is the node's value and max is the given max passed by its parent.

Code

```
public class CheckForBST
{

 /**
  * @param args
  */
 public static void main(String[] args)
 {
  Node a=new Node(12);
  Node b=new Node(3);
  Node c=new Node(15);
  Node d=new Node(10);
  Node e=new Node(14);
  Node f=new Node(17);
  Node g=new Node(4);
  Node h=new Node(11);
  Node i=new Node(5);
  a.left=b;
  a.right=c;
  b.right=d;
  c.left=e;
  c.right=f;
  d.left=g;
  d.right=h;
  g.right=i;
  System.out.println(isBST(a));

 }

 public static boolean isBST(Node root)
 {
  return checkBooleanAndReturn(root, Integer.MAX_VALUE,
    Integer.MIN_VALUE);
 }

 private static boolean checkBooleanAndReturn(Node root,
```

```
 int max,int min)
{
 if(root==null)return true;
 boolean leftOk=true;
 boolean rightOk=true;
 boolean centerOk=true;
 if(root.left!=null)
  leftOk=checkBooleanAndReturn(root.left,
   root.value,min);
 if(root.value>=min &&root.value<=max)
  centerOk=true;
 else
  centerOk=false;
 if(root.right!=null)
  rightOk=checkBooleanAndReturn(root.right,
   max,root.value);
 return leftOk&&centerOk&&rightOk;
}


static class Node
{
 public Node left;
 public Node right;
 public int value;
 public Node(int value)
 {
  this.value=value;
 }
}
}
```

Create tree from in and pre

Problem

Create a binary tree fromgiven inorder and preorder traversal.

Solution

The first node in the preorder traversal is the root. Now if we try to find the root node in the inorder array, all the elements to its left will be in left subtree of root and elements to its right will be in right subtree of root. We recursively call the function for its left and right and continuously increase the preorder index. As java does not support pass by reference, we will pass an single element array so that the value change can be reflected outside the function.

Code

```
public class CreateTreeFromInPre
{
 public static void main(String[] args)
 {
  int[] inorder =
  { 4, 2, 5, 1, 6, 3 };
  int[] preorder =
  { 1, 2, 4, 5, 3, 6 };
  Node root = buildBinaryTree(inorder, preorder);
  printNice(root);
 }

 public static Node buildBinaryTree(int[] inorder,
  int[] preorder)
 {
  return buildBinaryTree(inorder, 0, inorder.length,
    preorder, new int[]
  { 0 });
 }

 private static Node buildBinaryTree(int[] inorder,
  int inStartIndex, int inEndIndex, int[] preorder,
  int[] preIndex)
 {
  if (preIndex[0] == preorder.length)
   return null;
  Node root = new Node(preorder[preIndex[0]]);
  int findIndex = -1;
  for (int i = inStartIndex; i < inEndIndex; ++i)
  {
   if (preorder[preIndex[0]] == inorder[i])
   {
    findIndex = i;
   }
  }
  if (findIndex == -1)
```

```
   return null;
  preIndex[0]++;
  root.left = buildBinaryTree(inorder, inStartIndex,
   findIndex, preorder, preIndex);
  root.right = buildBinaryTree(inorder, findIndex + 1,
    inEndIndex, preorder, preIndex);
  return root;
 }

 static class Node
 {
  Node left;
  Node right;
  int value;

  public Node(int value)
  {
   this.value = value;
  }
 }

 public static void printNice(Node root)
 {
  if (root == null)
   return;
  else
  {
   System.out.print(root.value);
   if (root.left != null)
   {
    System.out.print("L->[");
    printNice(root.left);
    System.out.print("]");
   }
   if (root.right != null)
   {
    System.out.print("R->[");
    printNice(root.right);
    System.out.print("]");
   }
  }
 }
}
```

Find path of a node from root

Problem

Given a binary tree and one of its nodes, find the path from root to the given node

Solution

We solve this by recursion. We start from the root. If the current node is equal to the given node we add it to a list and return. In this way from any node we check the path returned by right and left children, any non null path contains the node.

Code

```
import java.util.ArrayList;
import java.util.List;

public class FindPathFromRoot
{
 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(6);
  Node g = new Node(7);
  Node h = new Node(8);
```

```
  a.left = b;
  a.right = c;
  b.left = d;
  c.left = e;
  c.right = f;
  f.left = g;
  f.right = h;

  List < node > path = findPath(a, e);
  for (Node node : path)
   System.out.println(node.value);
 }

 private static List < node > findPath(Node root, Node node)
 {
  if (root == null)
   return null;

  if (root == node)
  {
   List < node > path = new ArrayList < node >();
   path.add(root);
   return path;
  }
  List < node > leftPath = findPath(root.left, node);
  List < node > rightPath = findPath(root.right, node);
  if (leftPath != null)
  {
   leftPath.add(0, root);
   return leftPath;
  }
  if (rightPath != null)
  {
   rightPath.add(0, root);
   return rightPath;
  }
  return null;
 }

 static class Node
 {
  Node left;
  Node right;
  int value;

  public Node(int value)
  {
   this.value = value;
  }
 }
}
```

Find distance between two nodes in a binary tree

Problem

Find the distance between two nodes in a binary tree. Every node of the binary tree has left pointer, right pointer and value. There is no parent pointer given in the nodes. Given two nodes or node values of such a binary tree and root of the tree given, find the distance between the given nodes.

Solution

We will first find out the path of the two nodes from root using recursive path finding algorithm. Now we will traverse simultaneously along the two paths till we find a mismatch. Then we know the size of the two paths, so we can easily calculate the distance by the formula, length of path1 + length of path2 - 2*length of the common part.

Code

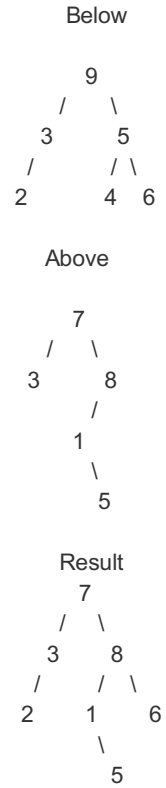
```
import java.util.ArrayList;
import java.util.List;

public class BinaryTreeDistanceBetweenNodes
{

 /**
  * @param args
  */
 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(6);
  Node g = new Node(7);
  Node h = new Node(8);
  a.left = b;
  a.right = c;
  b.left = d;
  c.left = e;
  c.right = f;
  f.left = g;
  f.right = h;

  int distance = findDistance(a, 2, 7);
  System.out.println(distance);

 }

 private static int findDistance(Node root, int val1, int val2)
 {
  List < node > path1 = new ArrayList < node >();
  List < node > path2 = new ArrayList < node >();
  findPath(root, val1, path1);
  findPath(root, val2, path2);
  if (path1.size() == 0 || path2.size() == 0)
   return -1;
  int index = 0;
  for (; index < path1.size(); ++index)
  {
   if (path1.get(index) != path2.get(index))
    break;
  }
  return path1.size() + path2.size() - 2 * index;
 }

 private static boolean findPath(Node root, int value, List < node > path)
 {
  if (root == null)
   return false;
  path.add(root);
  if (root.value == value)
  {
   return true;
  }
  if (findPath(root.left, value, path)
    || findPath(root.right, value, path))
   return true;
  path.remove(root);
  return false;
 }
```

```
 static class Node
 {
  Node left;
  Node right;
  int value;

  public Node(int value)
  {
   this.value = value;
  }
 }
}
```

Superimpose binary tree

Problem

Given two binary trees, find the resultant binary tree if we put one binary tree on top of another. For example,



Just look how the 2 and 6 are visible from the below tree and rest are covered by above tree.

Solution

We will do a preorder traversal along the two trees simultaneously, When above is null we will return the below node. when below is null we don't need to proceed further as the above tree will remain same for that subtree. So we return above when below is null. When above and below both are non null we will call the superimpose recursively to the left and right subtree.

Code

```
public class SuperImposeBinaryTree
{
 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(6);
```

```
 a.left = b;
 a.right = c;
 b.left = d;
 c.left = e;
 c.right = f;

 Node u = new Node(4);
 Node v = new Node(7);
 Node w = new Node(2);
 Node x = new Node(8);
 Node y = new Node(1);
 Node z = new Node(5);

 u.left = v;
 u.right = w;
 w.left = x;
 w.right = y;
 y.right = z;
 printNice(a);
 System.out.println();
 printNice(u);
 System.out.println();
 Node result = superImpose(a, u);
 printNice(result);
}

private static Node superImpose(Node below, Node above)
{
 if (above == null)
  return below;
 if (below == null)
  return above;
 above.left = superImpose(below.left, above.left);
 above.right = superImpose(below.right, above.right);
 return above;
}

public static void printNice(Node root)
{
 if (root == null)
  return;
 else
 {
  System.out.print(root.value);
  if (root.left != null)
  {
   System.out.print("L->[");
   printNice(root.left);
   System.out.print("]");
  }
  if (root.right != null)
  {
   System.out.print("R->[");
   printNice(root.right);
   System.out.print("]");
  }
 }
}

static class Node
{
 Node left;
 Node right;
 int value;

 public Node(int value)
 {
  this.value = value;
```

```
  }

  public static void printNice(Node root)
  {
   if (root == null)
    return;
   else
   {
    System.out.print(root.value);
    if (root.left != null)
    {
     System.out.print("L->[");
     printNice(root.left);
     System.out.print("]");
    }
    if (root.right != null)
    {
     System.out.print("R->[");
     printNice(root.right);
     System.out.print("]");
    }
   }
  }
 }
}
```

Does any path adds up to a given sum

Problem

Given a binary tree and a number, find out whether any path of the binary tree sums up to k. The binary tree nodes contain numbers. The path can be starting from anywhere, but it should not skip any node in its path. It should be continuous nodes along a path which sums up to k.

Solution

We try to solve it recursively, Suppose the given sum is k and the root of the binary tree is given. If k is equal to the value of the root node then we will return true. If the value is not equal to k then we will try to find the sum (k-value) in the left and right subtree by calling the same function. If anyone of those returns true, that means a path exists including the root which sums up to k. But the path may not start at the root, it can start anywhere. So along with the two calls with 'k-value' we call the left and right subtree with the original sum k also. If that returns true that means the path starts from somewhere else other than root.

Code

```
public class SumPossibleAlongPathBinaryTree
{
 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(6);
  Node g = new Node(7);
  a.left = b;
  a.right = c;
  b.left = d;
  c.left = e;
  c.right = f;
  f.left = g;

  boolean result = isSumPossibleAlongAnyPath(a, 18);
  System.out.println(result);
 }
```

```
 private static boolean isSumPossibleAlongAnyPath(Node a, int k)
 {
  return isSumPossibleAlongAnyPath(a, k, k);
 }

 private static boolean isSumPossibleAlongAnyPath(Node a, int k,int originalK)
 {
  if (k == 0)
   return true;
  if (a == null)
   return false;
  return isSumPossibleAlongAnyPath(a.left, k - a.value,originalK)
    || isSumPossibleAlongAnyPath(a.right, k - a.value,originalK)
    || isSumPossibleAlongAnyPath(a.left,originalK,originalK)
    || isSumPossibleAlongAnyPath(a.right, originalK,originalK);
 }

 static class Node
 {
  Node left;
  Node right;
  int value;

  public Node(int value)
  {
   this.value = value;
  }
 }

}
```

Remove duplicates from infinite integers

Problem

You are given an infinite stream of integers. The stream of integers is unsorted and are provided by iterator so that you don't have the access to all the elements at once. You have to return another iterator from the input iterator so that there are no duplicates and the input order is maintained.

Solution

We will use binary search tree to have a O(n log n) solution. Each number is inserted into an inner binary search tree and if no previous existence found, it is added to output stream. The output iterator always prepare the next element beforehand so that hasNext and next does not disagree. When the input iterator finishes so does the output. This has worst case complexity of O(n^2) for a sorted input sequence, we can improve this by any of the balanced search tree like red black tree.

Code

```
import java.util.Iterator;

public class RemoveDuplicateBST
{
 public static void main(String[] args)
 {
  System.out.println("Input\tOutput");
  Iterator inputIterator = new InputIterator(30);
  Iterator outputIterator = new OutputIterator(inputIterator);
  while (outputIterator.hasNext())
  {
```

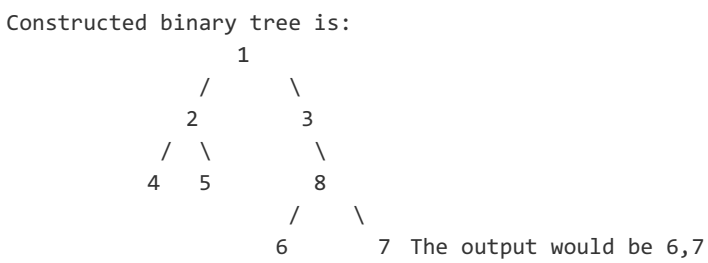
```
  System.out.println("\t<<" + outputIterator.next());
 }
}

private static class InputIterator implements Iterator
{
 int count;

 public InputIterator(int count)
 {
  this.count = count;
 }

 @Override
 public boolean hasNext()
 {
  return count != 0;
 }

 @Override
 public Integer next()
 {
  int input = (int) (Math.random() * 30);
  System.out.println(">>" + input);
  count--;
  return input;
 }

 @Override
 public void remove()
 {

 }
}

private static class OutputIterator implements Iterator
{
 Iterator inputIterator;
 Integer nextElement;
 Node root;

 public OutputIterator(Iterator inputIterator)
 {
  this.inputIterator = inputIterator;
  if (inputIterator.hasNext())
  {
   nextElement = inputIterator.next();
   if (nextElement != null)
    add(nextElement);
  }
 }

 private boolean add(int value)
 {
  return add(root, value);
 }

 private boolean add(Node current, int value)
 {
  Node node = new Node(value);
  if (current == null)
  {
   root = node;
   return true;
  }
  if (current.value == value)
  {
   return false;
```

```
    }
    if (current.value > value)
    {
     if (current.left == null)
     {
      current.left = node;
      return true;
     } else
     {
      return add(current.left, value);
     }
    } else
    {
     if (current.right == null)
     {
      current.right = node;
      return true;
     } else
     {
      return add(current.right, value);
     }
    }
   }

   @Override
   public boolean hasNext()
   {
    return nextElement != null;
   }

   @Override
   public Integer next()
   {
    int output = nextElement;
    nextElement = null;
    while (inputIterator.hasNext())
    {
     int input = inputIterator.next();
     if (add(input))
     {
      nextElement = input;
      break;
     }
    }
    return output;
   }

   @Override
   public void remove()
   {

   }
  }

  private static class Node
  {
   int value;
   Node left;
   Node right;

   public Node(int value)
   {
    this.value = value;
   }
  }
}
```

Find deepest node(s) of a binary tree

Problem

Given a binary tree find the node(s) at its deepest level. For example,



Solution

We do a preorder traversal of the tree. While calling the recursive function we pass an array containing level and list of nodes. During the preorder traversal, this same array is always passed. So the elements of this array work as global variables. Whenever we call the recursive function to a node's children we increase the level by 1. At each node if the level value in the array is less than the current level value then we clear the previous node list, as we found a deeper level. If the level value is equal then we keep on adding the current node in the list.

Code

```
import java.util.ArrayList;
import java.util.List;

public class FindDeepestNodes
{
 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(8);
  Node g = new Node(6);
  Node h = new Node(7);
  a.left = b;
  a.right = c;
  b.left = d;
  b.right = e;
  c.right = f;
  f.left = g;
  f.right = h;

  List nodes = findDeepestNodes(a);
  for (Node node : nodes)
  {
   System.out.print(node.value + ", ");
  }
 }

 @SuppressWarnings("unchecked")
 private static List findDeepestNodes(Node root)
 {
  Object[] levelNodes = new Object[2];
  levelNodes[0] = 0;
  levelNodes[1] = new ArrayList();
  findDeepestNodes(root, 1, levelNodes);
```

```
  return (List) levelNodes[1];
 }

 @SuppressWarnings("unchecked")
 private static void findDeepestNodes(Node root, int level,
   Object[] levelNodes)
 {
  if (root == null)
   return;
  if((Integer)levelNodes[0]<=level)
  {
   if((Integer)levelNodes[0] < level)
    ((List)levelNodes[1]).clear();
   levelNodes[0]=level;
   ((List)levelNodes[1]).add(root);
  }
  findDeepestNodes(root.left, level+1, levelNodes);
  findDeepestNodes(root.right, level+1, levelNodes);
 }

 static class Node
 {
  Node left;
  Node right;
  int value;

  public Node(int value)
  {
   this.value = value;
  }
 }

}
```

Maximum sum along any path of binary tree of positive numbers

Problem

Given a binary tree where each node contains positive numbers, find the maximum sum that is possible along any path of the binary tree.

Solution

We will solve this recursively. The function f(node) will return the maximum sum of the tree rooted at the node. So f(node)=node.value+ max(f(node.left),f(node.right)). f(node.left) and f(node.right) will be calculated recursively. For f(null) we will return 0;

Code

```
public class MaxSumTreePathPositive
{
 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(6);
  Node g = new Node(7);
  Node h = new Node(8);
```

```
  a.left = b;
  a.right = c;
  b.left = d;
  c.left = e;
  c.right = f;
  f.left = g;
  f.right = h;
  int maxSum = findMaxSum(a);
  System.out.println(maxSum);
 }

 private static int findMaxSum(Node root)
 {
  if (root == null)
   return 0;
  return root.value
    + Math.max(findMaxSum(root.left), findMaxSum(root.right));
 }

 static class Node
 {
  Node left;
  Node right;
  int value;

  public Node(int value)
  {
   this.value = value;
  }
 }

}
```

Maximum sum along any tree path with positive and negative values

Problem

You are given a binary tree with positive or negative numbers in its nodes. Find the maximum possible sum along any path of the binary tree.

Solution

We do a postorder traversal, so move from leaf to node. Along any path we keep on adding the value to a local sum. If local sum is less than zero we discard that path and make local sum to zero and keep on continuing up. Every time we find a local sum we compare it with a global maximum and update the global whenever local is greater than global. At any node we check the maximum local sum of left and right children and take the larger of these two values in our local sum calculation. When we reach a null node we make our local sum zero and return from there.

Code

```
public class MaxSumTreePathNegative
{
 public static void main(String[] args)
 {
  Node a = new Node(4);
  Node b = new Node(-2);
  Node c = new Node(-1);
  Node d = new Node(6);
  Node e = new Node(-5);
  Node f = new Node(-2);
  Node g = new Node(7);
```

```
  Node h = new Node(8);
  a.left = b;
  a.right = c;
  b.left = d;
  c.left = e;
  c.right = f;
  f.left = g;
  f.right = h;
  int maxSum = findMaxSum(a);
  System.out.println(maxSum);
 }

 private static int findMaxSum(Node root)
 {
  int[] localMax = new int [1];
  int[] globalMax = new int [1];
  findMaxSum(root, localMax, globalMax);
  return globalMax[0];
 }

 private static void findMaxSum(Node root, int[] localMax, int[] globalMax)
 {
  if (root == null)
  {
   localMax[0]=0;
   return;
  }
  localMax[0]=0;
  findMaxSum(root.left, localMax, globalMax);
  int leftLocal=localMax[0];
  localMax[0]=0;
  findMaxSum(root.right, localMax, globalMax);
  int rightLocal=localMax[0];
  int maxLocal=Math.max(leftLocal,rightLocal);
  if(maxLocal+root.value > 0)
  {
   localMax[0]=maxLocal+root.value;
   if(globalMax[0] < localMax [0])
    globalMax[0] = localMax [0];
  }
  else
  {
   localMax[0]=0;
  }
 }

 static class Node
 {
  Node left;
  Node right;
  int value;

  public Node(int value)
  {
   this.value = value;
  }
 }

}
```

Find kth smallest element in a Binary search tree

Problem

Given a binary search tree find the kth smallest element.

Solution

We will augment the binary search tree by storing the weight of left subtree rooted at a node. So any node will keep an extra count which is equal to the number of nodes in its left subtree. This augmentation is called order statistics tree. With the help of this augmentation we can find the kth smallest element in O(log n) expected complexity for a balanced binary search tree. Suppose we try to find the 6th smallest element, we start at root. If root has leftWeight value 3, that means there are only 3 elements that are smaller than root. So 6th smallest element cannot be on the left side of root. So we try to find the element in right subtree. While going to right subtree we now try to find 6-4=2nd smallest element, because we already had 3 smaller element in root's left subtree and root itself is smaller than the right subtree. So we call the recursive function on root.right. If the value of k is less than the leftWeight then we just go to the left subtree with the value k.

Code

```
public class KthLargestOnline
{
 public static void main(String[] args)
 {
  BST bst = new BST();
  int[] arr =
  { 12, 4, 5, 6, 2, 7, 8, 11, 2, 3 };
  for (int num : arr)
   bst.add(num);
  System.out.println(bst.getOrdered(4));
  arr = new int[]
  { 12, 1, 9, 14, 25 };
  for (int num : arr)
   bst.add(num);
  System.out.println(bst.getOrdered(6));

 }

 private static class BST
 {
  Node root;

  public void add(int num)
  {
   if (root == null)
   {
    root = new Node(num);
   } else
    add(root, num);
  }

  private void add(Node root, int num)
  {
   Node node = new Node(num);
   if (node.value < root.value)
   {
    root.leftWeight++;
    if (root.left == null)
     root.left = node;
    else
     add(root.left, num);
   } else
   {
    if (root.right == null)
     root.right = node;
    else
     add(root.right, num);
   }
  }
```

```
  public int getOrdered(int k)
  {
   return getOrdered(root, k);
  }

  private Integer getOrdered(Node root, int k)
  {
   if (root == null)
    return null;
   if (root.leftWeight > k)
   {
    return getOrdered(root.left, k);
   } else if (root.leftWeight < k)
   {
    return getOrdered(root.right, k - root.leftWeight);
   } else
   {
    return root.value;
   }
  }
 }

 private static class Node
 {
  int value;
  int leftWeight;
  Node left;
  Node right;

  public Node(int value)
  {
   this.value = value;
   this.leftWeight = 1;
  }
 }

}
```

Binary search tree with insertion order maintained

Problem

Design a binary search tree where the insertion order is maintained.

Solution

In the binary tree in each node maintains a next pointer which acts as a linked list through the nodes which maintain insertion order. There would be normal left and right pointer which maintains the property of the binary search tree. The tree will provide an additional method called getSortedOrder, which returns a list of elements in insertion order. After inserting every node we will add that new node to the linked list's last node and advance the last pointer to the next node. So any time we can insert a node in BST in O(log n) time and in O(1) we can add it to the linked list. If the BST allows deletion then to maintain the linked list in O(1) time we need a doubly linked list.

Code

```
import java.util.ArrayList;
import java.util.List;

public class BinaryTreeInsertionOrder
{
 public static void main(String[] args)
```

```
{
 BinaryTree bt = new BinaryTree();
 bt.add(4).add(5).add(2).add(9).add(14).add(6).add(3);
 List < integer > list = bt.getSortedOrder();
 for (Integer num : list)
  System.out.print(num + ", ");
 System.out.println();
 list = bt.getInsertionOrder();
 for (Integer num : list)
  System.out.print(num + ", ");
 System.out.println();
}

private static class BinaryTree
{
 Node root;
 Node head;
 Node last;

 public BinaryTree add(int num)
 {
  if (root == null)
  {
   root = new Node(num);
   head = root;
   last = root;
  } else
  {
   Node node = new Node(num);
   add(root, node);
   last.next = node;
   last = node;
  }
  return this;
 }

 private void add(Node root, Node node)
 {
  if (node.value < root.value)
  {
   if (root.left == null)
    root.left = node;
   else
    add(root.left, node);
  } else
  {
   if (root.right == null)
    root.right = node;
   else
    add(root.right, node);
  }

 }

 public List < integer > getInsertionOrder()
 {
  Node current = head;
  List < integer > list = new ArrayList < integer >();
  while (current != null)
  {
   list.add(current.value);
   current = current.next;
  }
  return list;
 }

 public List < integer > getSortedOrder()
 {
```

```
   List < integer > list = new ArrayList < integer >();
   inorder(root, list);
   return list;
  }

  private void inorder(Node root, List < integer > list)
  {
   if (root == null)
    return;
   inorder(root.left, list);
   list.add(root.value);
   inorder(root.right, list);
  }

 }

 private static class Node
 {
  int value;
  Node left;
  Node right;
  Node next;

  public Node(int value)
  {
   this.value = value;
  }

 }
}
```

Binary tree sum of nodes at odd levels

Problem

Given a binary tree find the sum of nodes present at odd levels. That means excluding nodes at every alternate levels.

Solution

We will a find the sum by recursively calling the function on left and right children of a node. While calling the function on any child node we will toggle a boolean flag to indicate whether this level is to include in the sum or not. If current level is to be included then add the current node's value to to the sum obtained from left and right children and return it. If the current level is not to be included then return only the sum of values returned from children.

Code

```
public class BinaryTreeSumOfValuesAtOddHeight
{
 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(8);
  Node g = new Node(6);
  Node h = new Node(9);
  a.left = b;
  a.right = c;
  b.left = d;
```

```
  b.right = e;
  c.right = f;
  f.left = g;
  f.right = h;
  int sum = findOddLevelSum(a);
  System.out.println(sum);
 }

 private static int findOddLevelSum(Node a)
 {
  return findOddLevelSum(a, true);
 }

 private static int findOddLevelSum(Node a, boolean oddLevel)
 {
  if (a == null)
   return 0;
  int childSum=findOddLevelSum(a.left, !oddLevel)
    + findOddLevelSum(a.right, !oddLevel);
  if (oddLevel)
   return a.value + childSum;
  else
   return childSum;
 }

 static class Node
 {
  Node left;
  Node right;
  int value;

  public Node(int value)
  {
   this.value = value;
  }
 }

}
```

Level order traversal without a queue

Problem

Given a binary tree print the node values in level order without using a queue.

Solution

We will use a map whose key is the level and value is the list of nodes. We will call a recursive function on the root passing a blank map and a level value of 1. At each node it will check whether the current level as a key is present in the map or not. If it is present it add the current node's value to the corresponding list. If the key is not present then it adds the level as key and a list containing current node as value. Then call the recursive function to its children with level value increased by one. When it hits null nodes, it returns. After this recursive method ends we have all the level wise data in the map. We will start from level 1 and print the data from its value list and keep on increasing the level till that level exists in the map. The space complexity for this solution is O(n) and time complexity is O(n).

Code

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
```

```
public class LevelOrderWithoutQueue
{

 /**
  * @param args
  */
 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(8);
  Node g = new Node(6);
  Node h = new Node(7);
  a.left = b;
  a.right = c;
  b.left = d;
  b.right = e;
  c.right = f;
  f.left = g;
  f.right = h;
  printLevelOrder(a);
 }

 private static void printLevelOrder(Node a)
 {
  HashMap < Integer, List < Integer > > levelMap =
   new HashMap < Integer, List < Integer > >();
  printLevelOrder(a, 1, levelMap);
  int level = 1;
  while (true)
  {
   List < Integer > list = levelMap.get(level);
   if (list == null)
    break;
   System.out.println(list);
   level++;
  }
 }

 private static void printLevelOrder(Node a, int level,
   HashMap < Integer, List < Integer > > levelMap)
 {
  if (a == null)
   return;
  List < Integer > list = levelMap.get(level);
  if (list == null)
  {
   list = new ArrayList < Integer > ();
   list.add(a.value);
   levelMap.put(level, list);
  } else
  {
   list.add(a.value);
  }
  printLevelOrder(a.left, level + 1, levelMap);
  printLevelOrder(a.right, level + 1, levelMap);
 }

 static class Node
 {
  Node left;
  Node right;
  int value;

  public Node(int value)
```

```
  {
   this.value = value;
  }
 }

}
```

Binary tree level with maximum number of nodes

Problem

You are given a binary tree. Print the level of the binary tree which has the maximum number of nodes. Break tie arbitrarily.

Solution

We will use a map to store the level wise nodes. The key will be the level and value will be list of nodes. We will call a function with a blank map and root node with level value 1. At any node if the level as key exist in the map we will add the node to the corresponding list or create a new key with the level and add a list containing the node as value. Then call the function recursively to its children with level value increased by 1. If a null node is hit then the function returns. When the recursive function returns the map will be filled with level wise data. In this map we will find the max length for any level and print accordingly.

Code

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

public class BinaryTreeLevelWithMaxNodes
{
 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(8);
  Node g = new Node(6);
  Node h = new Node(7);
  a.left = b;
  a.right = c;
  b.left = d;
  b.right = e;
  c.right = f;
  f.left = g;
  f.right = h;
  printMaxLevel(a);
 }

 private static void printMaxLevel(Node a)
 {
  HashMap < Integer, List < Integer > > levelMap =
    new HashMap < Integer, List < Integer > >();
  printLevelOrder(a, 1, levelMap);
  int level = 1;
  int maxNodes = 0;
  List < Integer > values = null;
  while (true)
  {
   List < Integer > list = levelMap.get(level);
   if (list == null)
```

```
    break;
   if (list.size() > maxNodes)
   {
    maxNodes = list.size();
    values = list;
   }
   level++;
  }
  System.out.println(values);
 }

 private static void printLevelOrder(Node a, int level,
   HashMap < Integer, List < Integer > > levelMap)
 {
  if (a == null)
   return;
  List < Integer > list = levelMap.get(level);
  if (list == null)
  {
   list = new ArrayList < Integer > ();
   list.add(a.value);
   levelMap.put(level, list);
  } else
  {
   list.add(a.value);
  }
  printLevelOrder(a.left, level + 1, levelMap);
  printLevelOrder(a.right, level + 1, levelMap);
 }

 static class Node
 {
  Node left;
  Node right;
  int value;

  public Node(int value)
  {
   this.value = value;
  }
 }

}
```

Find all wrong pairs in a BST

Problem

A binary tree is given. Which is supposed to be a BST, but it has some errors. Find all the pair of numbers which are not following the BST rule

Solution

In a BST all numbers in the left subtree of a node should be less than the node and all numbers in the right subtree of the node should be greater than the node. So given a node we can say what is the valid range of numbers in the left subtree and what is for the right. As long as the numbers in the node is in that range, it is fine. When some node breaks this range, then we print the pair of wrong numbers and create two ranges for its children. one range is to find the wrong pair considering its ancestors and another range is for a tree considering that wrong node as a root of a new BST.

Code

```
package dsalgo;

public class BinaryTreeWrongPairs {

  public static void main(String[] args) {
    Node a = new Node(5);
    Node b = new Node(10);
    Node c = new Node(14);
    Node d = new Node(6);
    Node e = new Node(3);
    a.left = b;
    a.right = c;
    b.left = d;
    c.right = e;
    findWrongPairsIn(a, null);
  }

  public static void findWrongPairsIn(Node root, Range range) {
    if (range == null) {
      range = new Range(Integer.MIN_VALUE, Integer.MAX_VALUE);
    }
    if (root == null) {
      return;
    }
    if (root.isNotIn(range)) {
      root.printTheWrongPairForThis(range);
      findWrongPairsIn(root.left, range);
      findWrongPairsIn(root.right, range);
    }
    findWrongPairsIn(root.left, root.newLeft(range));
    findWrongPairsIn(root.right, root.newRight(range));
  }
}


class Node {
  Node left;
  Node right;
  int value;

  public Node(int value) {
    this.value = value;
  }

  public boolean isNotIn(Range range) {
    if (value >= range.min && value <= range.max) {
      return false;
    }
    return true;
  }

  public Range newLeft(Range range) {
    int min = isNotIn(range) ? Integer.MIN_VALUE : range.min;
    return new Range(min, value);
  }

  public Range newRight(Range range) {
    int max = isNotIn(range) ? Integer.MAX_VALUE : range.max;
    return new Range(value, max);
  }

  public void printTheWrongPairForThis(Range range) {
    if (range.min > value) {
      System.out.println(range.min + "," + value);
    }
    if (range.max < value) {
      System.out.println(range.max + "," + value);
    }
```

```
  }
}


class Range {
  int min;
  int max;

  public Range(int min, int max) {
    this.min = min;
    this.max = max;
  }
}
```

Find depth of tree from parent array

Problem

A tree is represented by an array of integer where arr[i] represents the parent of ith node. arr[i]=-1 for root node. Find the depth of that tree

Solution

We can start with any node and calculate its depth by recursive function and then return the maximum depth among all the nodes. But in this process we will calculate the value of some nodes multiple times. So we will keep a depthArray which will keep track of the values for nodes that has been calculated once. Because of this memoization we will calculate each node exactly once. So the complexity is O(n)

Code

```
public class DepthFromParrentArray {
  public static void main(String[] args) {
    int[] parentArray = {3, -1, 3, 1, 5, 1, 0, 4, 2, 6};
    System.out.println(findDepth(parentArray));
  }

  private static int findDepth(int[] parentArray) {
    int[] depthArray = new int[parentArray.length + 1];
    for (int i = 0; i < parentArray.length; ++i) {
      findDepthRecursive(depthArray, parentArray, i);
    }
    return depthArray[parentArray.length];
  }


  private static int findDepthRecursive(int[] depth,
      int[] parentArray, int i) {
    if (i == -1) {
      return 0;
    }
    if (depth[i] != 0) {
      return depth[i];
    }
    depth[i] = 1 + findDepthRecursive(depth, parentArray,
        parentArray[i]);
    if (depth[i] > depth[parentArray.length]) {
      depth[parentArray.length] = depth[i];
    }
    return depth[i];
  }
}
```

Make BST from doubly linked list

Problem

You are given a sorted doubly linked list. Nodes of this doubly linked list have a left as previous and right as next pointer and a value. Convert this to a binary search tree using the same nodes.

Solution

We will search for the middle of this doubly linked list and make that middle node as the root of Binary search tree. Then we will recursively construct its left and right subtree.

Code

```
package dsalgo;

public class MakeBSTFromDoublyLinkedList {

 static class Node{
  Node left;
  Node right;
  int value;
  Node(int value){
   this.value=value;
  }
 }
 public static void main(String[] args) {
  Node a=new Node(1);
  Node b=new Node(2);
  Node c=new Node(3);
  Node d=new Node(4);
  Node e=new Node(5);
  Node f=new Node(6);
  Node g=new Node(7);
  Node h=new Node(8);
  Node i=new Node(9);
  Node j=new Node(10);

  a.right=b;
  b.right=c;
  c.right=d;
  d.right=e;
  e.right=f;
  f.right=g;
  g.right=h;
  h.right=i;
  i.right=j;

  j.left=i;
  i.left=h;
  h.left=g;
  g.left=f;
  f.left=e;
  e.left=d;
  d.left=c;
  c.left=b;
  b.left=a;
  Node root=makeBst(a);
 }

 public static Node makeBst(Node head){
  if(head==null)
   return null;
  if(head.right==null)
  return head;
  Node root=findMiddle(head);
  if(root.left!=null)
   root.left.right=null;
```

```
  root.left=makeBst(head);
  if(root.right!=null)
   root.right.left=null;
  root.right=makeBst(root.right);
  return root;
 }
 public static Node findMiddle(Node head){
  if(head.right==null)
  return head;
  Node firstPointer=head;
  while(firstPointer!=null){
   head=head.right;
   firstPointer=firstPointer.right;
   if(firstPointer==null)break;
   firstPointer=firstPointer.right;
   if(firstPointer==null)break;
   firstPointer=firstPointer.right;
  }
  return head;
 }
}
```

Draw a binary tree with ASCII

Problem

Write a program to draw a binary tree with fixed width fonts. Represent left links with a '/' and right link with '\'. For example like this.

```
      1
     / \
    /   \
   2     \
  / \     3
 4   5   / \
        9   \
            8
           / \
          6   7
```

Solution

Use the post order traversal of the binary tree to know the width required to represent the subtree rooted at a particular node. The width required is equal to sum of its left child's width, right child's width and its own text width. Recursively find the complete tree's width. Then start with the root with an x coordinate of width/2. Every printable strings are collected as a list of (x, y, string) format. Now each line of the drawing is represented by the elements in the list which have the same y values. For every line a string is created with required spaces such that the x coordinate values of the individual strings are obtained. And we print these strings from smaller to greater
values of y

Code

```
package com.dsalgo;

import java.util.ArrayList;
import java.util.List;
import java.util.TreeMap;

public class BinaryTreeNicePrint {

 public static void main(String[] args) {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
```

```
 Node e = new Node(5);
 Node f = new Node(8);
 Node g = new Node(6);
 Node h = new Node(7);
 Node i = new Node(9);
 a.left = b;
 a.right = c;
 b.left = d;
 b.right = e;
 c.left = i;
 c.right = f;
 f.left = g;
 f.right = h;
 nicePrint(a);
}

public static void nicePrint(Node root) {
 List< StringPoint > result = getStrings((getWidth(root) + 1) / 2, 0, root);
 TreeMap< Integer, List< StringPoint > > lines = new TreeMap<  >();
 for (StringPoint s : result) {
  if (lines.get(s.y) != null) {
   lines.get(s.y).add(s);
  } else {
   List< StringPoint > l = new ArrayList<  >();
   l.add(s);
   lines.put(s.y, l);
  }
 }
 for (List< StringPoint > l : lines.values()) {
  System.out.println(flatten(l));
 }
}

private static String flatten(List< StringPoint > l) {
 int x = 0;
 StringBuilder sb = new StringBuilder();
 for (StringPoint s : l) {
  sb.append(new String(new char[s.x - x]).replace('\0', ' '));
  sb.append(s.value);
  x = sb.length();
 }
 return sb.toString();
}

private static int getWidth(Node root) {
 int width = 0;
 if (root.left != null) {
  width += getWidth(root.left);
 }
 if (root.right != null) {
  width += getWidth(root.right);
 }
 width += ("" + root.value).length();
 return width;
}

private static List< StringPoint > getStrings(int x, int y, Node root) {
 List< StringPoint > result = new ArrayList< StringPoint >();
 result.add(new StringPoint(x - ("" + root.value).length() / 2, y, ""
   + root.value));
 if (root.left != null) {
  int width = getWidth(root.left);
  int i = 0;
  for (; i <  (width + 1) / 2; ++i)
   result.add(new StringPoint(x - i - 1, y + i + 1, "/"));
  result.addAll(getStrings(x - i - 1, y + i + 1, root.left));
 }
 if (root.right != null) {
```

```
   int width = getWidth(root.right);
   int i = 0;
   for (; i <  (width + 1) / 2; ++i)
    result.add(new StringPoint(x + i + 1, y + i + 1, "\\"));
   result.addAll(getStrings(x + i + 1, y + i + 1, root.right));
  }
  return result;
 }

 static class StringPoint {
  Integer x;
  Integer y;
  String value;

  StringPoint(int x, int y, String value) {
   this.x = x;
   this.y = y;
   this.value = value;
  }

  @Override
  public String toString() {
   return "(" + x + "," + y + "," + value + ")";
  }
 }

 static class Node {
  Node left;
  Node right;
  int value;

  public Node(int value) {
   this.value = value;
  }
 }
}
```

Remove duplicate subtree

Problem

Compact a binary tree by removing the duplicate subtrees. For example the binary tree given in figure 1 has two duplicate subtrees. The duplicate parts are circled in figure 2. Wherever such duplication occurs, we should remove that duplicate subtree and point the link to the existing subtree as described in figure 3 1

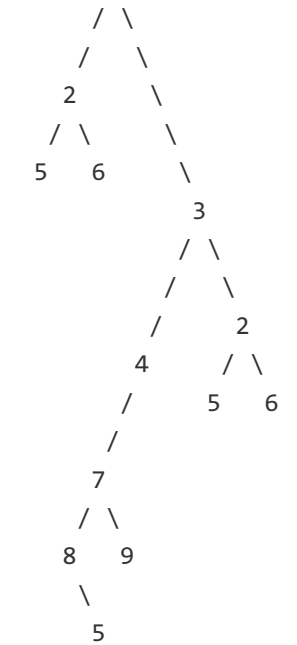


Figure 1

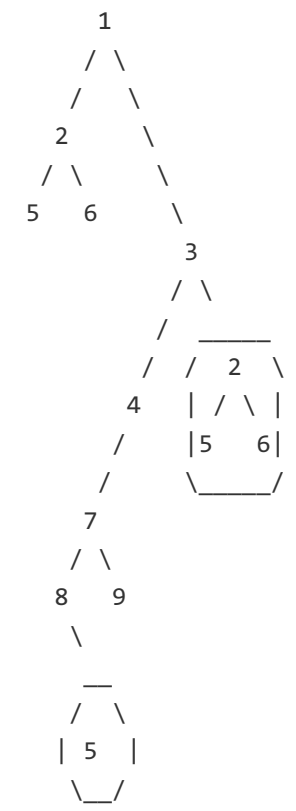


Figure 2

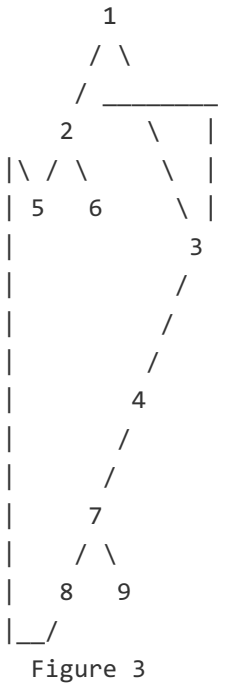


Figure 3

Solution

First we need to figure out how to determine the equality of two subtrees. As we know we can construct a binary tree decisively from its preorder and inorder representations, we will use this property for equality of two subtrees. We construct a NodeHash

object corresponding to every nodes of a binary tree. A NodeHash object contains a preorder list and inorder list of the subtree rooted at a node. We override the equals and hashCode of NodeHash object so that if two NodeHash objects contain exact same preorder and inorder lists, they are equal themselves. With this property being set, we can use the NodeHash object as a map key. Now while traversing the tree we keep on checking and storing the NodeHash objects as key and Node objects as value in a map. Whenever we find that the NodeHash object is equal to an already present key, that means we have found a duplicate subtree. Whenever we find a duplicate NodeHash, we get the corresponding Node object from the map and replace in the tree. If we don't find it, we insert the new NodeHash and recursively carry on with other nodes of the tree.

Code

```
package com.dsalgo;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class CompactBinaryTree {

 public static void main(String[] args) {

  Node a = new Node(1);
  Node b = new Node(2);
  Node bb = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node ee = new Node(5);
  Node eee = new Node(5);
  Node f = new Node(6);
  Node ff = new Node(6);
  Node g = new Node(7);
  Node h = new Node(8);
  Node i = new Node(9);
  a.left = b;
  b.left = e;
  b.right = f;
  a.right = c;
  c.left = d;
  c.right = bb;
  bb.left = ee;
  bb.right = ff;
  d.left = g;
  g.left = h;
  g.right = i;
  h.right = eee;
  compactTree(a);
  System.out.println("Let's see if it worked...");
  System.out.println("left child of a: "
    + System.identityHashCode(a.left));
  System.out.println("right child of c: "
    + System.identityHashCode(c.right));
  System.out.println("right child of h: "
    + System.identityHashCode(h.right));
  System.out.println("left child of b: "
    + System.identityHashCode(b.left));
 }

 static void compactTree(Node root) {
  Map< NodeHash, Node > map = new HashMap<  >();
  compactTree(root, map);
 }

 static void compactTree(Node root, Map< NodeHash, Node > map) {
  NodeHash nodeHash = new NodeHash(root);
  map.put(nodeHash, root);
  if (root.left != null) {
   NodeHash leftHash = new NodeHash(root.left);
   if (map.containsKey(leftHash)) {
```

```
    root.left = (Node) map.get(leftHash);
   } else {
    compactTree(root.left, map);
   }
  }
  if (root.right != null) {
   NodeHash rightHash = new NodeHash(root.right);
   if (map.containsKey(rightHash)) {
    root.right = (Node) map.get(rightHash);
   } else {
    compactTree(root.right, map);
   }
  }
 }

 static class Node {
  private Node left;
  private Node right;
  private int value;

  public Node(int value) {
   this.value = value;
  }

  @Override
  public int hashCode() {
   return value;
  }

  @Override
  public boolean equals(Object obj) {
   return value == ((Node) obj).value;
  }

  @Override
  public String toString() {
   return "" + value;
  }
 }

 static class NodeHash {
  List< Node > preOrderList;
  List< Node > inOrderList;

  NodeHash(Node root) {
   preOrderList = preOrder(root);
   inOrderList = inOrder(root);
  }

  @Override
  public boolean equals(Object obj) {
   NodeHash n = (NodeHash) obj;
   return preOrderList.equals(n.preOrderList)
     && inOrderList.equals(n.inOrderList);
  }

  @Override
  public int hashCode() {
   return 31 * preOrderList.hashCode() + inOrderList.hashCode();
  }

  List< Node > preOrder(Node root) {
   List< Node > result = new ArrayList<  >();
   if (root == null)
    return result;
   result.add(root);
   result.addAll(preOrder(root.left));
   result.addAll(preOrder(root.right));
```

```
   return result;
  }

  List< Node > inOrder(Node root) {
   List< Node > result = new ArrayList<  >();
   if (root == null)
    return result;
   result.addAll(inOrder(root.left));
   result.add(root);
   result.addAll(inOrder(root.right));
   return result;
  }
 }
}
```

Running Median

Problem

Find the running median after entering every number from a sequence of infinite integers.

Brute force

If we keep each number in a sorted sequence then cost of single entry is O(n) and finding median is O(n). A slight modification can be done by keeping the middle pointer and adjusting it based on the insertion on its left side and right side. In that case finding median after insertion is O(1). But the overall cost for finding median still remains O(n) as insertion in sorted sequence is necessary after each number is entered.

Better solution

We can keep two heaps which divides the entered number in two almost equal halves. Half of the number would be greater than the median and the rest would be lesser. The upper half will be maintained in a min heap and the lower half will be maintained in a max heap. In this arrangement we can find out in O(1) time whether a new number would go to the upper half or lower half. All we need to do is to compare the new number with the head of two heaps. After deciding we can insert in a heap in O(log n) time. After this insertion if the heaps are unbalanced, we can just move from one heap to another. which is again of O(log n) complexity. And now we can find the median in O(1) time. If two heaps contain same number of elements then median is the average of the head of two heaps. If one is greater, then median is the head of the larger heap.

Further discussion

Whenever the answer of some problem requires knowing greater among something or lesser among something but it does not matter whether the rest of the elements are ordered or not, heap might give the optimal solution.

Code

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Random;


public class RunningMedian
{
    PriorityQueue<Integer> upperQueue;
    PriorityQueue<Integer> lowerQueue;

    public RunningMedian()
    {
        lowerQueue=new PriorityQueue<Integer>(
          20,new Comparator<Integer>()
        {

            @Override
            public int compare(Integer o1, Integer o2)
            {

                return -o1.compareTo(o2);
            }

        });
        upperQueue=new PriorityQueue<Integer>();
        upperQueue.add(Integer.MAX_VALUE);
        lowerQueue.add(Integer.MIN_VALUE);
    }

    public double getMedian(int num)
    {
        //adding the number to proper heap
            if(num>=upperQueue.peek())
                upperQueue.add(num);
            else
                lowerQueue.add(num);
        //balancing the heaps
        if(upperQueue.size()-lowerQueue.size()==2)
            lowerQueue.add(upperQueue.poll());
        else if(lowerQueue.size()-upperQueue.size()==2)
            upperQueue.add(lowerQueue.poll());
```

```
        //returning the median
        if(upperQueue.size()==lowerQueue.size())
            return(upperQueue.peek()+lowerQueue.peek())/2.0;
        else if(upperQueue.size()>lowerQueue.size())
            return upperQueue.peek();
        else
            return lowerQueue.peek();

    }
    public static void main(String[] args)
    {
        Random random=new Random();
        RunningMedian runningMedian=new RunningMedian();
        System.out.println("num\tmedian");
        for(int i=0;i<50;++i)
        {
            int num=random.nextInt(100);
            System.out.print(num);
            System.out.print("\t");
            System.out.println(
              runningMedian.getMedian(num));
        }

    }

}
```

Maximum k integers using min heap

Problem

Find k maximum integers from an array of infinite integers.

Brute force

We can sort the integers and return the top k elements. The complexity is O(n log n) because of sorting. As n is very large we would have to store all the elements in memory for sorting them.

Better solution

We can implement a min heap to hold exactly k elements. Every time a new number is added we can check with the top of the element which is the minimum of the current k elements. If new number is less than the min heap top then that number is discarded otherwise it is added in the heap after removing the current top. In this way we always keep only k items in memory and the complexity is O(n log k) (assumption is that k is significantly lesser than n)

Code

```
import java.util.PriorityQueue;

public class MaxKUsingMinHeap
{
 public static void main(String[] args)
 {
  int[] arr =
  { 3, 46, 2, 56, 3, 38, 93, 45, 6, 787, 34, 76,
    44, 6, 7, 86, 8, 44, 56 };
  int[] result = getTopElements(arr, 5);
  for (int i : result)
  {
   System.out.print(i + ",");
  }
 }

 public static int[] getTopElements(int[] arr, int k)
 {
  PriorityQueue<Integer> minHeap =
   new PriorityQueue<Integer>();
  for (int i = 0; i < arr.length; ++i)
  {
   int currentNum = arr[i];
   if (minHeap.size() < k)
    minHeap.add(currentNum);
   else if (currentNum > minHeap.peek())
   {
    minHeap.poll();
    minHeap.add(currentNum);
   }
  }
  int[] result = new int[minHeap.size()];
  int index = 0;
  while (!minHeap.isEmpty())
  {
   result[index++] = minHeap.poll();
  }
  return result;
```

```
 }
}
```

Max heap and BST in same structure

Problem

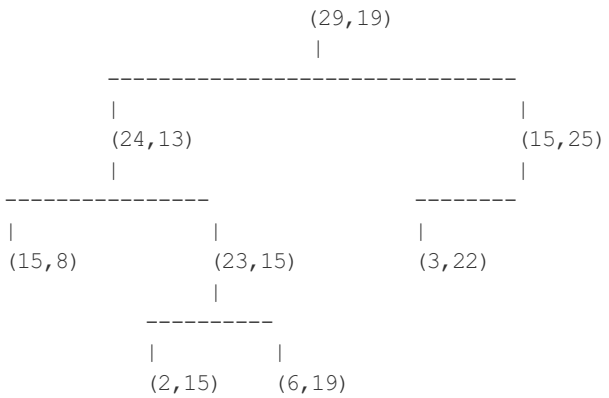
Some number pairs are given. Arrange the number pairs in the form of a binary tree in such a fashion so that the first numbers in each tuple is arranged in a max heap and the second number in each tuple is arranged in a binary search tree.

Each node of the binary tree will contain a tuple, when we look at the first number of each node the structure will look as a max heap, when we look at the second number of each node the structure will look as binary search tree.

Example

Input:
(2,15),(6,19),(3,22),(24,13),(29,19),(23,15),(15,25),(15,8)

Output:



Solution

Suppose each node has two values named heapValue and treeValue. Among all the nodes we first try to find the highest heapValue. This node will have to be the root of the tree for making it a max heap. Now we look at the treeValue of this node. All the nodes having lesser tree value than this will go to the left subtree and rest will go to the right subtree. Which can be built recursively.

Code

```
import java.util.ArrayList;
import java.util.List;


public class MaxHeapAndBinarySearchTree
{
 public static void main(String[] args)
 {
  List < node > list = new ArrayList < node >();
  for (int i = 0; i < 10; ++i)
  {
   list.add(new Node((int) (30 * Math.random()),
     (int) (30 * Math.random())));
  }
  Node root=createHeapAndBST(list);
  printNice(root);
 }

 private static Node createHeapAndBST(List < node > list)
 {
  if(list.size()==0)
   return null;
```

```
 Node top=list.get(0);
 for(Node node:list)
 {
  if(node.heapValue>top.heapValue)
   top=node;
 }
 list.remove(top);
 List < node > leftList=new ArrayList < node >();
 List < node > rightList=new ArrayList < node >();
 for(Node node:list)
 {
  if(node.treeValue<=top.treeValue)
   leftList.add(node);
  else
   rightList.add(node);
 }
 top.left=createHeapAndBST(leftList);
 top.right=createHeapAndBST(rightList);
 return top;
}

static class Node
{
 public Node left;
 public Node right;
 public int heapValue;
 public int treeValue;

 public Node(int heapValue, int treeValue)
 {
  this.heapValue = heapValue;
  this.treeValue = treeValue;
 }
}

public static void printNice(Node root)
{
 if (root == null)
  return;
 else
 {
  System.out.print("(" + root.heapValue + "," + root.treeValue + ")");
  if (root.left != null)
  {
   System.out.print("L->[");
   printNice(root.left);
   System.out.print("]");
  }
  if (root.right != null)
  {
   System.out.print("R->[");
   printNice(root.right);
   System.out.print("]");
  }
 }
}
```

Merge N sorted arrays

Problem

Given n sorted arrays. Merge them into a single sorted array.

Solution

While merging two sorted sequences we always look at the front element and pick the lowest of them. Then the next element from the picked array takes place of the previous element and the process continues till one of them is empty. In case of N arrays if we do the same process, the time complexity of checking the front elements is O(n). If there are N arrays all having elements of the order of N. Then there are O(N^2) elements and for every element we will have to find the minimum in O(n) time. So the overall complexity will be O(N^3). We can decrease the complexity to O(N^2 log N) by adding the arrays to a heap. Each node of the heap will be an array whose priority will be decided by the front element. So if we pick the root of the heap we will find the smallest of the front elements of all the arrays. After adding it to the resultant array we will remove it and again insert the array into the heap, This process will take O(log N). So the overall complexity will become O(N^2 log N)

For implementing this, we created a class ArrayContainer which will hold the array inside an object. And whose natural ordering will be decided by the front element of the array. So when all the array containers are inserted into the heap, top element will be the array with smallest front element.

Code

```
/*
For problem and solution description please visit the link below
http://www.dsalgo.com/2013/02/merge-n-sorted-arrays.html
*/
package com.dsalgo;
import java.util.PriorityQueue;


public class MergeNSortedArrays
{
 public static void main(String[] args)
 {
  int[]arr1={2,4,6,8,9,12,14,16};
  int[]arr2={3,6,7,9,22,25,28};
  int[]arr3={2,5,7,8,10,11,16};
  int[]arr4={4,8,23,26,28};
  int[]result=mergeNArrays(new int[][]{arr1,arr2,arr3,arr4});
  for(int i:result)
  {
   System.out.print(i+",");
  }
 }

 private static int[] mergeNArrays(int[][] sortedArrays)
 {
  int totalLength=0;
  PriorityQueue < ArrayContainer > heap=new PriorityQueue < ArrayContainer >();

  for(int i=0;i < sortedArrays.length;++i)
  {
   totalLength+=sortedArrays[i].length;
   heap.add(new ArrayContainer(sortedArrays[i]));
  }
  int[]result=new int[totalLength];
  int index=0;
  while(!heap.isEmpty())
  {
   ArrayContainer arrayContainer=heap.poll();
   result[index++]=arrayContainer.getNextInt();
   if (arrayContainer.isEmpty())
    continue;
   heap.add(arrayContainer);
  }
  return result;
 }
 private static class ArrayContainer implements Comparable < ArrayContainer >
 {
  private int startIndex;
  private int[]array;
  public ArrayContainer(int[]array)
  {
```

```
   this.array=array;
   startIndex=0;
  }
  public boolean isEmpty()
  {
   return startIndex==array.length;
  }
  public int peek()
  {
   return array[startIndex];
  }
  public int getNextInt()
  {
   return array[startIndex++];
  }
  @Override
  public int compareTo(ArrayContainer o)
  {
   return new Integer(peek()).compareTo(o.peek());
  }

 }
}
```

Find order of letters

Problem

Find the lexicographic order of letters from given lexicographic order of words.

Problem variation

A stone scripture has been found from some historical ruins. Archeologists guessed it is a dictionary and the words are ordered based on some predefined order of letters. But there is no direct evidence of which letter comes after what. Write an algorithm to find out the ordering of the letters from the ordering of the words.

Solution

From the ordering of any two words we can find ordering of a letter. For example if cat comes before dog, then we can say that c is before d. If cab comes before cat, we can say b comes before t. In this way from n words we can find n*(n-1)/2 combinations of words. From which we can find at most n*(n-1)/2 relations. We can represent these relations as edges of a directed graph where vertices are individual letters.
We represent the graph as an incidence matrix. Any other representation will also do. Now if we do topological sorting of this directed graph, we will get the ordering of the vertices. Which is nothing but the ordering of the letters. For topological sorting we try to find out a vertex which has no inbound edges. That is the vertex with least ordering. Then we remove all the outbound edges from that vertex and continue the process by finding next vertex with no inbound edges. In this way we find the complete ordering of the letters. During the process at any time if we can't find any vertex with zero inbound edge, that means there is a conflict in the ordering of the words and hence the ordering of the letters can't be done.

Code

```
import java.util.HashMap;

public class FindOrderOfLetter
{
 public static void main(String[] args)
 {
  String[] words =
  { "car", "cat", "cbr", "deer", "egg", "god",
    "rabe", "race", "rat", "tar" };
  char[] letters = getLetterOrdering(words);
  if (letters == null)
   System.out.println("not possible");
  else
  {
   for (char ch : letters)
    System.out.print(ch + ",");
  }
 }

 private static char[] getLetterOrdering(String[] words)
 {
  HashMap<Character, Integer> characters =
   new HashMap<Character, Integer>();
  for (String word : words)
  {
   for (int i = 0; i < word.length(); ++i)
   {
    char character = word.charAt(i);
    if (!characters.keySet().contains(character))
    {
     characters.put(character, characters.size());
    }
   }
  }
  boolean[][] adjacency = new boolean[characters.size()]
```

```
   [characters.size()];
  for (int i = 0; i < words.length - 1; ++i)
  {
   for (int j = i + 1; j < words.length; ++j)
   {
    String prevWord = words[i];
    String nextWord = words[j];
    for (int k = 0; k < Math.min(prevWord.length(),
      nextWord.length()); ++k)
    {
     char prevCharacter = prevWord.charAt(k);
     char nextCharacter = nextWord.charAt(k);
     if (prevCharacter != nextCharacter)
     {
      adjacency[characters.get(prevCharacter)][characters
        .get(nextCharacter)] = true;
      break;
     }
    }
   }
  }

  char[] result = new char[characters.size()];
  int resultIndex = 0;
  while (!characters.isEmpty())
  {
   char lowChar = ' ';
   for (Character nextCharacter : characters.keySet())
   {
    int nextIndex = characters.get(nextCharacter);
    boolean lowest = true;
    for (Character prevCharacter : characters.keySet())
    {
     int prevIndex = characters.get(prevCharacter);
     if (adjacency[prevIndex][nextIndex])
     {
      lowest = false;
      break;
     }
    }
    if (lowest)
    {
     lowChar = nextCharacter;
     result[resultIndex++] = nextCharacter;
     break;
    }
   }
   if (lowChar == ' ')
   {
    return null;
   } else
   {
    characters.remove(lowChar);
    lowChar = ' ';
   }
  }
  return result;
 }

}
```

Find longest interviewer chain

Problem

In an organization employees are interviewed with one or more previous employees. Those previous employees are also interviewed by some other employees during their joining. In this way an interview chain is formed, where every node in the chain has been interviewed by the next node in the chain. You are given such information of interviewers for every employees. Given any employee in the organization find the longest such chain possible.

Solution

Every employee maintains a list of interviewers. We recursively find the longest chain. We call the same function to find the longest

chain of all of its interviewers. Whichever path has the highest length, we append the current employee to that list and return. If the employee has no interviewers then the recursion ends there and we return a single element list which contain only that particular employee.

Code

```
import java.util.ArrayList;
import java.util.List;

public class InterviewHierarchy
{
 public static void main(String[] args)
 {
  Employee ceo = new Employee("ceo");
  Employee director1 = new Employee("director1");
  Employee director2 = new Employee("director2");
  Employee director3 = new Employee("director3");
  Employee manager1 = new Employee("manager1");
  Employee manager2 = new Employee("manager2");
  Employee manager3 = new Employee("manager3");
  Employee employee1 = new Employee("employee1");
  Employee employee2 = new Employee("employee2");
  Employee employee3 = new Employee("employee3");
  director1.addInterviewer(ceo);
  director2.addInterviewer(ceo).addInterviewer(director1);
  director3.addInterviewer(director1).addInterviewer(director2);
  manager1.addInterviewer(ceo).addInterviewer(director1)
    .addInterviewer(director2).addInterviewer(director3);
  manager2.addInterviewer(director2).addInterviewer(director3)
    .addInterviewer(manager1);
  manager3.addInterviewer(director1).addInterviewer(director2)
    .addInterviewer(manager2).addInterviewer(manager1);
  employee1.addInterviewer(director1).addInterviewer(director2)
    .addInterviewer(manager3).addInterviewer(manager2);
  employee2.addInterviewer(manager3).addInterviewer(manager2)
    .addInterviewer(manager1).addInterviewer(employee1);
  employee3.addInterviewer(employee1).addInterviewer(employee2)
    .addInterviewer(director3).addInterviewer(manager2);

  List result = findLongestInterviewChain(employee2);
  System.out.println("Longest interview chain of " + employee2.getName());
  for (Employee employee : result)
   System.out.print(employee.getName() + ", ");
 }

 private static List findLongestInterviewChain(Employee employee)
 {
  int maxLength = -1;
  List longestPath = new ArrayList();
  for (Employee interviewer : employee.getInterviewers())
  {
   List path = findLongestInterviewChain(interviewer);
   if (path.size() > maxLength)
   {
    maxLength = path.size();
    longestPath = path;
   }
  }
  longestPath.add(0, employee);
  return longestPath;
 }

 private static class Employee
 {
  private List interviewers;
  private String name;
```

```
  public Employee(String name)
  {
   this.name = name;
   interviewers = new ArrayList();
  }

  public String getName()
  {
   return name;
  }

  public List getInterviewers()
  {
   return interviewers;
  }

  public Employee addInterviewer(Employee employee)
  {
   this.interviewers.add(employee);
   return this;
  }
 }
}
```

Find local minima in an array

Problem

Given an array of unique integers whose first two numbers are decreasing and last two numbers are increasing, find a number in the array which is local minima. A number in the array is called local minima if it is smaller than both its left and right numbers. For example in the array 9,7,2,8,5,6,3,4
2 is a local minima as it is smaller than its left and right number 7 and 8. Similarly 5 is another local minima as it is between 8 and 6, both larger than 5. You need to find any one of the local minima.

Solution

First we need to understand that if in an array of unique integers first two numbers are decreasing and last two numbers are increasing there ought to be a local minima. Why so? We can prove it in two ways. First we will do it by negation. If first two numbers are decreasing, and there is no local minima, that means 3rd number is less than 2nd number. otherwise 2nd number would have been local minima. Following the same logic 4th number will have to be less than 3rd number and so on and so forth. So the numbers in the array will have to be in decreasing order. Which violates the constraint of last two numbers being in increasing order. This proves by negation that there need to be a local minima.

We can prove this in some other way also. Suppose we represent the array as a 2-D graph where the index of the numbers in the array represents the x-coordinate. and the number represents the y-coordinate. Now for the first two numbers, derivative will be negative, and for last two numbers derivative will be positive. So at some point the derivative line will have to cross the x axis. As the array contains only unique elements there cannot be a derivative point on the x axis. Because that will mean that two consecutive index having same number. So for any intersection of x axis by the derivative line will be a local minima.

We will solve this problem in O(log n) time by divide and conquer method. We will first check the mid index of the array. If it is smaller than its left and right, then it is the answer. If it is bigger than the left number then from start to left we have a subproblem, and as we proved already that starting with decreasing and ending with increasing sequence array will have to have a local minima, we can safely go to the left subarray. Otherwise if mid is bigger than its right, then we go to the right subarray. This way we guarantee a O(log n) algorithm to find any of the local minima present in the array.

Code

```
public class LocalMinima
{
```

```
 public static void main(String[] args)
 {
  int[] arr = {64, 14, 52, 27, 71, 19, 63, 1, 16, 57};
  for (int num : arr)
   System.out.print(num + ", ");
  System.out.println();
  int minima = findLocalMinima(arr);
  System.out.println(minima);
 }

 private static int findLocalMinima(int[] arr)
 {
  return findLocalMinima(arr, 0, arr.length);
 }

 private static int findLocalMinima(int[] arr, int start, int end)
 {
  int mid = (start + end) / 2;
  if (mid - 2 < 0 && mid + 1 >= arr.length)
   return -1;
  if (arr[mid - 2] > arr[mid - 1] && arr[mid - 1] < arr[mid])
   return arr[mid - 1];
  if (arr[mid - 1] > arr[mid - 2])
   return findLocalMinima(arr, start, mid);
  else
   return findLocalMinima(arr, mid, end);
 }

}
```

Separate words in sentence

Problem

Given a valid sentence without any spaces between the words and a dictionary of valid english words, find the individual words in the sentence. For example, "therearesome" -> "there are so me" "somewordshiden" -> "so me words hi den"

Solution

We start scanning the sentence from left. As we find a valid word we need to check whether the rest of the sentence can make valid words or not. Because in some situations the first found word from left side can leave a remaining portion which is not further separable. So in that case we should come back and leave the current found word and keep on searching for the next word. And this process is recursive because to find out whether the right portion is separable or not, we need the same logic. So we will use recursion and backtracking to solve this problem. To keep track of the found words we will use a stack. Whenever the right portion of the string does not make valid words, we pop the top string from stack and continue finding.

Code

```
import java.util.HashSet;
import java.util.Set;
import java.util.Stack;

public class SeparateWordsInSentence
{
 public static void main(String[] args)
 {
  String sentence = "therearesomewordshiddenhere";
  String[] dictionary =
  { "the", "a", "i", "here", "so", "hid", "there", "are",
    "some", "word", "words", "hid", "hi", "hidden", "he",
    "here", "her", "rear", "me", "den" };
  String[] words = getSeparatedWords(sentence, dictionary);
  for (String word : words)
   System.out.println(word);

 }

 private static String[] getSeparatedWords(String sentence,
   String[] dictionary)
 {
  Set<String> validWords = new HashSet<String>();
  for (String validWord : dictionary)
   validWords.add(validWord);
  Stack<String> words = new Stack<String>();
  if (isSeparable(sentence, validWords, 0, words))
  {
   return words.toArray(new String[] {});
  }
```

```
  return null;
 }

 private static boolean isSeparable(String sentence,
  Set<String> validWords,
   int startIndex, Stack<String> foundWords)
 {
  if (startIndex == sentence.length())
   return true;
  boolean hasWord = false;
  for (int i = startIndex + 1; i <= sentence.length(); ++i)
  {
   String currentSubstring = sentence.substring(startIndex, i);
   if (validWords.contains(currentSubstring))
   {
    foundWords.push(currentSubstring);
    if (isSeparable(sentence, validWords, i, foundWords))
    {
     hasWord = true;
     break;
    }
    foundWords.pop();
   }
  }
  if (!hasWord)
   return false;
  return true;
 }
}
```

First increasing second decreasing tuple

Problem

Given N pairs of integers, write an algorithm to sort the pairs so that the first number in each pair is sorted increasingly while the second number in each pair is sorted decreasingly. The first and second numbers in each pair can be swapped. Sometimes there will be no solution, in that case return an error.

Examples:

```
*
* Input: 1 5 7 1 3 8 5 6
*
* Output: 1 7 <- Swapped 1 5 6 5 <- Swapped 8 3 <- Swapped
*
* Input: 1 5 6 9
*
* Output: Not Possible
```

Solution

This is solved by backtracking. In a loop we try to put each tuple at the beginning and then try to put the consecutive tuples following the constraint, that first number will be increasing and second number will be decreasing. Every tuple is tried two ways one with the given tuple and another one is with reversed. If we can put a tuple following the constraint we call the recursive function with the rest of the tuples, If we cannot put a tuple we try with the next tuple. If no tuple can be put at that location we backtrack and try with another tuple in the previous location. If we are able to put the tuple when there is only one tuple left, then we found a solution. If we backtrack to the first element and could not find a solution after trying with all tuples, then we return the error that no arrangement possible.

Code

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class IncreasingDecreasingTuple
{
 static int count = 0;
```

```
/**
 * Given N pairs of integers, write an algorithm to sort
 * the pairs so that the first number in each pair is sorted
 * increasingly while the second number in each pair is sorted
 * decreasingly. The first and second numbers in each pair
 * can be swapped. Sometimes there will be no solution,
 * in that case return an error.
 *
 * Examples:
 *
 * Input: 1 5 7 1 3 8 5 6
 *
 * Output:
 * 1 7  < - Swapped 1 5 6 5  < - Swapped 8 3  < - Swapped
 *
 * Input: 1 5 6 9
 *
 * Output: Not Possible
 *
 * @param args
 */
public static void main(String[] args)
{
 Integer[][] input =
 {
 { 1, 5 },
 { 7, 1 },
 { 3, 8 },
 { 5, 6 } };
 List < List < Integer >  >  list =
   new ArrayList < List < Integer >  > ();
 for (Integer[] tuple : input)
 {
  List < Integer >  tupleList =
   new ArrayList < Integer > ();
  for (Integer el : tuple)
  {
   tupleList.add(el);
  }
  list.add(tupleList);
 }

 System.out.println(getArrangeMent(list));

}

public static List < List < List < Integer >  >  >
  getArrangeMent(List < List < Integer >  >  list)
{
 List < List < List < Integer >  >  >  result =
  new ArrayList < List < List < Integer >  >  > ();
 for (int i = 0; i  <  list.size(); ++i)
 {
  List < Integer >  tuple = list.get(i);
  List < List < Integer >  >  subList =
   new ArrayList < List < Integer >  > (list);
  subList.remove(i);
  List < List < List < Integer >  >  >  r1 =
   getRecurseArrangeMent(tuple, subList);
  if (r1 != null)
   result.addAll(prepend(tuple, r1));
  List < Integer >  rTuple = new ArrayList < Integer > (tuple);
  Collections.reverse(rTuple);
  List < List < List < Integer >  >  >  r2 =
   getRecurseArrangeMent(rTuple, subList);
  if (r2 != null)
   result.addAll(prepend(rTuple, r2));
```

```
 }
 return result;
}

public static List < List < List < Integer >  >  >
  getRecurseArrangeMent(
   List < Integer >  start, List < List < Integer >  >  list)
{
 List < List < List < Integer >  >  >  result =
   new ArrayList < List < List < Integer >  >  > ();
 if (list.size() == 0)
  return result;

 for (int i = 0; i  <  list.size(); ++i)
 {
  List < Integer >  tuple = list.get(i);
  if (tuple.get(0)  >= start.get(0) && start.get(1)  >=
   tuple.get(1))
  {
   List < List < Integer >  >  subList =
    new ArrayList < List < Integer >  > (list);
   subList.remove(i);
   List < List < List < Integer >  >  >  r1 =
    getRecurseArrangeMent(tuple, subList);
   if (r1 != null)
    result.addAll(prepend(tuple, r1));
  }
  List < Integer >  rTuple = new ArrayList < Integer > (tuple);
  Collections.reverse(rTuple);
  if (rTuple.get(0)  >= start.get(0) && start.get(1)  >=
   rTuple.get(1))
  {
   List < List < Integer >  >  subList =
    new ArrayList < List < Integer >  > (list);
   subList.remove(i);
   List < List < List < Integer >  >  >  r1 =
    getRecurseArrangeMent(rTuple, subList);
   if (r1 != null)
    result.addAll(prepend(rTuple, r1));
  }
 }
 if (result.size() != 0)
  return result;
 return null;
}

public static List < List < List < Integer >  >  >
 prepend(List < Integer >  start,
  List < List < List < Integer >  >  >  list)
{
 List < List < List < Integer >  >  >  results =
  new ArrayList < List < List < Integer >  >  > ();
 if (list.size() == 0)
 {
  List < List < Integer >  >  result =
   new ArrayList < List < Integer >  > ();
  result.add(start);
  results.add(result);
 } else
 {
  for (List < List < Integer >  >  result : list)
  {
   result.add(0, start);
   results.add(result);
  }
 }
 return results;
}
```

```
}
```

Two numbers sum up to k

Problem

Given a sorted integer array and an integer k, find two numbers which sum upto k.

Solution

We put two pointers at two ends of the array. As the array is sorted, if the sum is less than k, then by moving left pointer right side will increase the sum. If the sum exceeds k, we can move the right pointer to the left. The complexity is O(n). This is possible, because the array is sorted, we will discuss some other way in our next program where this complexity can be reached when the array is not sorted.

Code

```
public class TwoSumToK
{
 public static void main(String[] args)
 {
  int[]arr={1,2,3,4,5,6,7,8,9,12,13,14,16,32,44};
  printIndexesForSumK(arr, 16);
 }

 public static void printIndexesForSumK(int[]arr,int k)
 {
  int startIndex=0;
  int endIndex=arr.length-1;
  while(startIndex<endIndex)
  {
   int sum=arr[startIndex]+arr[endIndex];
   if(sum==k)
   {
    System.out.println(arr[startIndex]+","+arr[endIndex]);
    startIndex++;
    endIndex--;
   }
   else if (sum<k)
    startIndex++;
   else if(sum>k)
    endIndex--;
  }
 }

}
```

Two numbers sum up to k unsorted

Problem

Given an integer array and an integer k, find two numbers which sum upto k.

Solution

In this unsorted array we can apply the previous algorithm by sorting the array. But that would make our complexity atleast O(n log n). As the best sorting would run in O(n log n) time. We can make the complexity O(n) by using O(n) space. We will store the number in a hashset. and search for the number k-n in the hashset. Whenever the match is found, that means the sum of k is possible by these two numbers. As expected complexity of finding the number in the hashset is O(1) for n such operations total complexity would become O(n)

Code

```
import java.util.HashSet;


public class UnsortedTwoSumToK
{
 public static void main(String[] args)
 {
  int[]arr={6,7,8,9,1,16,2,3,14,13,4,5,12,32,44};
  printIndexesForSumK(arr, 16);
 }

 public static void printIndexesForSumK(int[]arr,int k)
 {
  HashSet<Integer> hashSet=new HashSet<Integer>();
  for(int num:arr)
  {
   if(hashSet.contains(k-num))
    System.out.println(num+","+(k-num));
   hashSet.add(num);
```

```
  }
 }

}
```

Any numbers sum up to k iterative

Problem

Given an integer array and a number k. Find out whether it is possible to have a sum k by using any number of numbers from the array.

Solution

We solve this by dynamic programming. In an iterative bottom up approach. Suppose we have a two dimensional boolean array named memo. Where memo [i][j] is true if the sum j is possible by using array elements less than i. So if i=0, that means no array element is being used for calculating the sum. So memo[0][j] is always false; now we have two for loops one for the array elements and one for the weights. In every step of the iteration our problem depends on the previous subproblem. Suppose our array element is 3 and required sum is 5. There might be two cases, this element 3 is either included in the solution or not. If it is included then we need to find whether the sum 2 is possible with the elements before it. If it is not included then we need to find whether the sum 5 is possible with the elements before it. If any one of these two results are true then the sum 5 is possible with the array elements upto that point. So our final result will be memo[array length][given k]. As this value will depend on the previous array elements, we start from 0,0 and progress from there. As all elements are integer, the value of k can vary from 0 to k in steps of 1.

Code

```
public class AnyNumberSumUptoK
{

 public static void main(String[] args)
 {
  int[]arr={4,6,2,8};
  System.out.println(isSumPossible(arr,22));
 }

 public static boolean isSumPossible(int[] arr, int k)
 {
  boolean[][] memo=new boolean[arr.length+1][k+1];
  for(int i=1;i<=arr.length;++i)
  {
   for(int w=1;w<=k;++w)
   {
    if(w==arr[i-1])
     memo[i][w]=true;
    else if(w>arr[i-1])
     memo[i][w]=memo[i-1][w]||memo[i-1][w-arr[i-1]];
    else
     memo[i][w]=memo[i-1][w];
   }
  }
  return memo[arr.length][k];
 }
}
```

Any numbers sum up to k recursive

Problem

Given an integer array and a number k. Find out whether it is possible to have a sum k by using any number of numbers from the array.

Solution

We will apply dynamic programming to solve this problem. Let us have a function func(index,sum) which gives us true if the sum is possible by adding any number from an array using numbers starting from index till the end. So func(0,k) means whether the sum k is possible by using all the elements of the given array. So this is our original problem. So we need to find out func(0,k). In this function we try to decide whether the number at the index position is required to make the sum k or not. If it is needed that means we need to make the number k-arr[index] with the remaining numbers to its right side. If it is not needed that means we need to make the number k with the remaining numbers. If any one of this is true that means it is possible to have k as a sum with the elements of the array. In this solution we take a recursive approach, that is we start from func(0,n) and recursively call the other functions in top down approach. We fill the memo array to reuse the value of a function so that for next iteration with the same values the function doesn't need to recompute. So this way we span from top to bottom and then fill the array from bottom to top to produce the final result. There is another approach which is iterative and don't need recursion where we directly fill the array from bottom to top. We will discuss that in our next page.

Code

```
public class AnyNumberSumUptoKRecursive
{
 public static void main(String[] args)
 {
  int [] arr={3,5,6,2};
  System.out.println(isSumPossible(arr, 13));
 }

 public static boolean isSumPossible(int[]arr, int k)
 {
  Boolean [][]memo=new Boolean[arr.length+1][k+1];
  return isSumPossible(memo,arr,0,k);
 }

 private static boolean isSumPossible(
```

```
  Boolean[][]memo,
  int[]arr,int i,int k)
 {
  if(memo[i][k]!=null)
   return memo[i][k];
  if(i==arr.length-1)
  {
   if(arr[i]==k||k==0)
    return true;
   else
    return false;
  }
  if(arr[i]>k)
  {
   memo[i][k]=isSumPossible(memo,arr,i+1,k);
   return memo[i][k];
  }
  memo[i][k]=isSumPossible(memo,arr,i+1,k-arr[i])||
   isSumPossible(memo,arr, i+1, k);
  return memo[i][k];
 }

}
```

Container loading recursive

Problem

There are some containers with different weights. A ship can hold maximum W weight. The problem is to find out what is the maximum we can load this container using the containers so that it does not exceeds W

Problem variation

An integer array is given, and an integer k is given. Find out the maximum sum possible using the elements of the array which does not exceed k

Solution

This is similar to the previous problem of finding out whether we can make a sum of k using the numbers. But here instead of trying to make k as the sum, we try to reach as near as possible to the number k but not more than k. In a similar fashion we will use dynamic programming to solve this problem. We will device a function which takes to key parameters index and sum and returns the maximum sum possible not exceeding sum. And using the items from index to end of the array. So at every recursion we have two choice whether to include the index-th element in the optimal solution or not. Suppose our function signature is func(index,sum), then func(0,k) returns the final answer. So at each step if the index-th element is included then the final sum becomes element+func(index+1,sum-element). Whereas if it is not included then the sum is func(index+1,sum). The functions examine these two values by calling recursive functions and returns the greater of these two. To minimize the complexity we keep a two dimensional integer array which is initialized after computing the value of func. So that later call to the function with same value does not need recomputation. In this way we calculate only once per cell of the two dimensional array. As the array dimensions are n and k, the complexity is O(nk).

Code

```
public class MaximumSumUptoKRecursive
{
 public static void main(String[] args)
 {
  int[]arr={4,6,3,9};
  System.out.println(getMaxSum(arr, 14));
 }
 public static int getMaxSum(int[]arr,int k)
 {
  Integer[][]memo=new Integer[arr.length+1][k+1];
  return getMaxSum(memo,arr,0,k);
 }
 private static int getMaxSum(Integer[][]memo,int[]arr,int i,int k)
 {
  if(i==arr.length)
  {
   return 0;
  }
  if(memo[i][k]!=null)
  {
   return memo[i][k];
  }
  if(arr[i]>k)
  {
   memo[i][k]=getMaxSum(memo,arr, i+1, k);
   return memo[i][k];
  }
  memo[i][k]=Math.max(getMaxSum(memo,arr,i+1,k),
   getMaxSum(memo,arr,i+1,k-arr[i])+arr[i]);
  return memo[i][k];
 }

}
```

Longest palindrome dynamic

Problem

Given a string find the longest possible substring which is a palindrome.

Solution

We will use dynamic programming to solve this problem. Given a substring if the first and last character are not same then it is not a palindrome, but if the end characters are same then the solution depends upon the inner string. So this is a subproblem, to find out whether the inner string is a palindrome or not. This subproblem may need to be solved several times. So for this we store the result of once computed substring in a two dimensional array, where array[i][j]=true off substring(i,j) is a palindrome. So the base cases are arr[i][i]=true, arr[i][i+1]=true if character at i and i+1 is same. For other cases arr[i][j]=arr[i-1][j-1] AND (charAt(i)==charAt(j)). While computing the two dimensional array we keep track of highest found length and startIndex. The complexity of this solution is O(n^2) because we will fill each of the cells of 2 d array only once.

Code

```
public class LargestPalindromeRecursive
{
 public static void main(String[] args)
 {
  String str = "acbcaccccaccc";
  String result = findLargestPalindrome(str);
  System.out.println(result);
 }

 private static String findLargestPalindrome(String str)
 {
  if(str==null ||str.length()==0)
   return "";
  boolean[][] memo = new boolean[str.length()][str.length()];
  int maxStart = 0;
  int maxLength = 1;
  for (int startIndex = 0; startIndex < str.length();
   ++startIndex)
  {
   memo[startIndex][startIndex] = true;
  }
  for (int startIndex = 0; startIndex < str.length() - 1;
    ++startIndex)
  {
   if (str.charAt(startIndex) == str.charAt(startIndex + 1))
   {
    memo[startIndex][startIndex + 1] = true;
    maxStart = startIndex;
    maxLength = 2;
   }
  }
  for (int length = 3; length <= str.length(); ++length)
  {
   for (int startIndex = 0; startIndex < str.length() -
    length + 1; ++startIndex)
   {
    int endIndex = startIndex + length - 1;
    if (str.charAt(startIndex) == str.charAt(endIndex)
      && memo[startIndex + 1][endIndex - 1] == true)
    {
     memo[startIndex][endIndex] = true;
     maxStart = startIndex;
     maxLength = length;
    }

   }
  }
  return str.substring(maxStart, maxStart + maxLength);
 }
```

```
}
```

Longest common subsequence

Problem

Given two strings find their longest common subsequence. A subsequence is a sequence which can be derived by deleting some of the elements of the original sequence. For example. ale is a subsequence of apple. gre is a subsequence of greed. If a string is subsequence of two strings, i,e it can be obtained by removing some characters from two strings then it is called a common subsequence. The problem is to find a subsequence of highest length of two given strings.

Solution

We will solve this by dynamic programming. If two strings end with the same character then we can remove that character from both the strings and compute the longest common subsequence for the rest of the string and at the end append the removed character to the result. If they don't end in the same character then one of the end characters will not be in the longest common subsequence. The longest common subsequence is maximum of LCS(A[0,n] , B[0,N-1]) and LCS(A[0,N-1] , B[0,N]). These two assumptions give the following formulas
LCS(a[0,n] , b[0,n]) = a[n] + LCS(a[0,n-1] , b[0,n-1]) when a[n] = b[n]
= Max(LCS(a[0,n] , b[0,n-1]) , LCS(a[0,n-1] , b[0,n]) when a[n] != b[n]

We solve this by memoizing the result in a m * n array. Each of the cell is computed once, so the complexity is O(mn). For two strings of length n the order is O(n^2)

Code

```
public class LongestCommonSubsequence
{
 public static void main(String[] args)
 {
  String a = "alfkjalfjlkj";
  String b = "ajflaklfjlaj";
  String result = findLCS(a, b);
  System.out.println(result);
 }

 private static String findLCS(String a, String b)
 {
  int[][] memo = new int[a.length() + 1][b.length() + 1];

  for (int i = a.length() - 1; i >= 0; --i)
   for (int j = b.length() - 1; j >= 0; --j)
   {
    if (a.charAt(i) == b.charAt(j))
     memo[i][j] = memo[i + 1][j + 1] + 1;
    else
     memo[i][j] = Math.max(memo[i + 1][j], memo[i][j + 1]);
   }

  int i = 0;
  int j = 0;

  StringBuffer result = new StringBuffer();
  while (i < a.length() && j < b.length())
  {
   if (a.charAt(i)==b.charAt(j))
   {
    result.append(a.charAt(i));
    i++;
    j++;
   } else if (memo[i+1][j] > memo[i][j+1])
    i++;
   else
    j++;
  }
  return result.toString();
```

```
  }
}
```

Gold coins in pots game

Problem

Some pots filled with gold coins and arranged in a line. There are different number of gold coins in different pots. The number is mentioned on each of the pots. Two players will take turn alternatively. In each turn a player can pick any one the two pots at the two ends of the line. Find the winning strategy for a player getting chance to play first, assuming the opponent player is also playing optimally. How many coins will the first player get if he plays optimally?

Solution

Suppose the pots are arranged in L[0] to L[n]. First player can pick either L[0] or L[n]. If he picks L[0] Second player can pick either L[1] or L[n]. If first picks L[n], second player can pick L[0] or L[n-1]. As the second player is also playing optimally he will also use the same strategy as the first player. Let us assume f(i,j) is the number of coins a player would get if he plays optimally from a point when two ends are i and j (0<=i<=j<=n). So f(0,n) would be the number of coins that first player would get after playing optimally. As second player is also playing optimally, he will pick a pot such that after his picking first player will get fewer gold if played optimally. If second player takes pot number 1, player 1 will get f(2,n). If second player takes pot number n, first player will get f(1,n-1). Second player will play such that 1st player gets minimum of these two options. So if first player takes pot 0 his final count of gold will be L[0]+Min(f(2,n),f(1,n-1)). Similarly if he picks the last pot this total count will be L[n]+Min(f(0,n-2),f(1,n-1)). From these two items whichever is greater First player should pick that, and that should be his winning strategy as playing first.

This is implemented with dynamic programming as we can see optimal subproblem and recurring subproblem. A 2 dimensional array is used to memoize the function output. So L[i][j]=f(i,j). We will initialize the array elements only once. So the complexity will be O(n^2).

Code

```
public class PotsOfGoldGame
{
 public static void main(String[] args)
 {
  int[] goldPots =
  { 12, 32, 4, 23, 6, 42, 16, 3, 85, 23, 4, 7, 3, 5, 45, 34, 2, 1 };
  int coins = getMaxGold(goldPots);
  System.out.println(coins);
 }

 private static int getMaxGold(int[] goldPots)
 {
  Integer[][]memo=new Integer[goldPots.length][goldPots.length];
  return getMaxGold(goldPots, 0, goldPots.length - 1,memo);
 }

 private static int getMaxGold(int[] goldPots, int startIndex, int endIndex,Integer[][]memo)
 {
  if (startIndex > endIndex)
   return 0;
  if(memo[startIndex][endIndex]!=null)
   return memo[startIndex][endIndex];
  int coinsIfStart = goldPots[startIndex]
    + Math.min(getMaxGold(goldPots, startIndex + 2, endIndex,memo),
      getMaxGold(goldPots, startIndex + 1, endIndex - 1,memo));
  int coinsIfEnd = goldPots[endIndex]
    + Math.min(getMaxGold(goldPots, startIndex, endIndex - 2,memo),
      getMaxGold(goldPots, startIndex + 1, endIndex - 1,memo));
  memo[startIndex][endIndex]=Math.max(coinsIfStart, coinsIfEnd);
  return memo[startIndex][endIndex];
 }
}
```

Array equal sum partition

Problem

Partition an integer array in two parts such that sum of each parts are same. The elements need not to be contiguous.

Brute force

If we try to find out all the subsets of the set it is of the order 2^n. For each of this subset we will have to find the sum and check it with the half of the total sum. So the total complexity will become O(n*2^n)

Better Solution

The sum of two parts should be equal. That means sum should be equal to half of total sum. As they are all integers so if the sum is odd number then it is not possible to partition. If they are even we will try to find whether that half of sum is possible by adding numbers from the array. We solve this by dynamic programming. We take a two dimensional array L of size count+1, sum/2+1. Where L[i,j]=maximum sum possible with elements of array 0 to i and sum not exceeding j. We compute this array from L[0,0] to L[n,sum/2]. So for computing L[i,j] we need to check the element i. if it is greater than j then it cannot be included in the sum so L[i,j]=L[i-1,j]. If ith element is less than j, then we should consider two cases and take the greater of them. First case is where we don't include the element in sum. Then L[i,j] = L[i-1,j] which is exactly like before. Now if we include the element, then L[i,j]=a[i]+L[i-1,j-a[i]]. In this way we fill this 2-d array. If L[n,sum/2] is equal to sum/2 that means the array can be partitioned in two parts whose sums are equal. Then we find out which elements have been included in that sum by backtracking through that 2-d array.

For backtracking through the array we just apply the reverse logic. If L[i,j] is equal to a[i]+L[i-1,j-a[i]] then that element is included in the partition. Then we continue this process to L[i-1,j-a[i]). If it is not equal then the element is not included in the final sum, so we proceed to L[i-1,j] After we get the complete list of elements in a partition we remove them from the original array to get the second partition.

Code

```
import java.util.ArrayList;
import java.util.List;

public class EqualSumPartition
{
 public static void main(String[] args)
 {
  Integer[] arr ={ 1, 2, 3, 6, 4, 5, 7 };
  System.out.println("Original Array");
  for (Integer num : arr)
  {
   System.out.print(num + ", ");
  }
  System.out.println();

  Integer[][] parts = partition(arr);
  if (parts == null)
  {
   System.out.println("partition not possible");
   return;
  }
  System.out.println("partition");
  for (Integer num : parts[0])
   System.out.print(num + ", ");
  System.out.println();
  for (Integer num : parts[1])
   System.out.print(num + ", ");
 }

 private static Integer[][] partition(Integer[] arr)
 {
  List < integer > list = new ArrayList < integer >();
  List < integer > part = new ArrayList < integer >();

  int sum = 0;
  for (Integer num : arr)
  {
   list.add(num);
```

```
    sum += num;
   }
   if (sum % 2 == 1)
    return null;
   sum /= 2;
   int[][] memo = new int[arr.length + 1][sum + 1];
   for (int i = 1; i <= arr.length; ++i)
   {
    for (int s = 1; s <= sum; ++s)
    {
     if (arr[i - 1] > s
       || memo[i - 1][s] > arr[i - 1]
         + memo[i - 1][s - arr[i - 1]])
      memo[i][s] = memo[i - 1][s];
     else
     {
      memo[i][s] = arr[i - 1] + memo[i - 1][s - arr[i - 1]];
     }
    }
   }
   if (memo[arr.length][sum] != sum)
    return null;
   int i = arr.length;
   int s = sum;
   while (s > 0 && i > 0)
   {
    if (arr[i - 1] <= s
      && memo[i][s] == arr[i - 1] + memo[i - 1][s - arr[i - 1]])
    {
     part.add(arr[i - 1]);
     s = s - arr[i - 1];
    }
    i--;
   }
   for (Integer num : part)
    list.remove(num);
   Integer[][] result = new Integer[2][];
   result[0] = part.toArray(new Integer[] {});
   result[1] = list.toArray(new Integer[] {});
   return result;
  }

}
```

Find a subset with given average

Problem

An integer array is given and an average value is given. Find a subset of the array which has the given average. For example input array is 1,3,4,5 and average is 2.5 then the subset {1,4} is the answer as its average is 2.5

Solution

We use dynamic programming to solve this problem. Suppose L is a 3-d boolean array. L[i,j,k]=true if using array element up to index i, a sum of j is possible using k number of elements from that range, and false otherwise. If we don't take the ith element then L[i,j,k]=L[i-1,j,k]. If we take the ith element then L[i,j,k]=L[i-1,j-arr[i],k-1]. Based on this formula we fill the 3-d array from bottom up. So Given the array, sum and count this array can tell whether a sum can be attained using k elements. If an average is given we do this operation n times where n is the count of the elements in the array. So if average is a, we search for L[n,a,1], L[n,2*a,2], ..., L[n,n*a,n]. If anyone of these searches returns true then we can obtain the given average. The function finds whether that sum is attainable with given count and returns the subset.

To return the subset we backtrack the 3-d array. If L[i,j,k]=L[i-1,j,k] then ith element should not be in the subset. If L[i,j,k]=L[i-1,j-arr[i],k-1] then that element is present in the subset.

The complexity of the solution is O(n^4). O(n^3) for filling the 3-d array and then doing this process n times for finding average.

Code

```
import java.util.ArrayList;
import java.util.List;

public class IsAveragePossible
{
 public static void main(String[] args)
 {
  int[] arr =
  { 1, 4, 5, 3, 8 };
  double average = 3.5;
  for (int i = 1; i <= arr.length; ++i)
  {
   double sum = i * average;
   if (sum == (int) sum)
   {
    List < integer > result = isSumPossible(arr, (int) sum, i);
    if (result != null)
    {
     for (int num : result)
      System.out.print(num + ", ");
     System.out.println();
     return;
    }
   }
  }

  System.out.println("Not possible");
 }

 private static List < integer > isSumPossible(int[] arr, int sum, int count)
 {
  boolean[][][] memo = new boolean[arr.length + 1][sum + 1][count + 1];
  memo[0][0][0] = true;

  for (int i = 1; i <= arr.length; ++i)
   for (int j = 1; j <= sum; ++j)
    for (int k = 1; k <= count; ++k)
    {
     if (k == 1 && j == arr[i - 1])
      memo[i][j][k] = true;
     else if (arr[i - 1] > j)
      memo[i][j][k] = memo[i - 1][j][k];
     else
      memo[i][j][k] = memo[i - 1][j][k]
        || memo[i - 1][j - arr[i - 1]][k - 1];
    }
  if (memo[arr.length][sum][count] == false)
   return null;
  int i = arr.length;
  int j = sum;
  int k = count;
  List < integer > list = new ArrayList < integer >();
  while (i > 0 && j > 0 && k > 0)
  {
   if (memo[i][j][k] == memo[i - 1][j][k])
   {

   } else if (k == 1 && j == arr[i - 1])
   {
    list.add(arr[i - 1]);
    break;
   } else if (arr[i - 1] <= j
     && memo[i][j][k] == memo[i - 1][j - arr[i - 1]][k - 1])
```

```
   {
    j = j - arr[i - 1];
    k = k - 1;
    list.add(arr[i - 1]);
   }
   i--;
  }
  return list;
 }
}
```

Share price max profit recursive

Problem

You are given the prices of a given stock for some days. You were allowed to own at most one stock at any time and you must not hold any stock at the end of that period. You can do at most k transactions. Find out the maximum profit that you could have.

Solution

At any given day, if we own the stock we have two choices, either we can sell it or we can keep it. if we sell it, we have a subproblem of finding out the max profit with the rest of the days with one less transaction. If we don't sell it we have a subproblem of finding out the max profit with the rest of the days with the same number of transaction. We find out the maximum of this two scenarios. Similarly if we don't own the share at any given day we have two options, we can buy it or not.

Code

```
package dsalgo;

public class StockPriceMaxProfitRecursive {

 public static void main(String[] args) {
  int[] prices = { 400, 402, 435, 398, 378, 400, 432, 432, 402 };
  System.out.println(getMaxProfit(prices, 0, 7, false));
 }

 public static int getMaxProfit(int[] prices, int startDay,
   int maxTransaction, boolean hasStock) {
  if (maxTransaction == 0 || startDay == prices.length)
   return 0;
  if (hasStock) {
   return Math.max(
     getMaxProfit(prices, startDay + 1, maxTransaction, true),
     getMaxProfit(prices, startDay + 1, maxTransaction - 1,
       false) + prices[startDay]);
  } else {
   return Math.max(
     getMaxProfit(prices, startDay + 1, maxTransaction, false),
     +getMaxProfit(prices, startDay + 1, maxTransaction - 1,
       true) - prices[startDay]);
  }
 }

}
```

Increasing array subsequence

Problem

You are given an integer array. Find all the subsequences of the array which has the elements in increasing order. B is a subsequence of array A if B can be formed by removing some elements from the array A without disturbing the order of elements.

For example {2,5,6,1,3} is the input array, Then {2,5}, {2,6}, {2,6,1}, {2,1,3} are some of its subsequences. In these {2,5}, {2,6} are increasing subsequences.

Solution

We will use memoization to solve this problem. At any index i of the given array we will have a subproblem of finding all the increasing subsequences till the index i. Then we will check with i+1th element. Some of these subsequences maintain the increasing property after adding this element. we will add this i+1th element to those subsequences and add the number itself as a single element subsequence to the previously memoized solution. As this i+1th element can itself be starting of another sequence. The complexity of the algorithm is output sensitive as we have to find all possible outputs. Let there be k number of increasing subsequences possible. Then the complexity is O(n*k). The term k is the total possible subsequeces. Which is of the order of 2^n. So in upper bound term it is O(n*2^n). But this complexity will only arises when all the subsequences are increasing subsequences. That occurs when the given array is sorted increasingly. In a random array the value of k is supposed to be much less than 2^n.

Code

```
import java.util.ArrayList;

public class IncreasingArraySubsequence
{

 /**
  * create subsequence of a given array where every
  * element in the subsequence is greater than its
  * previous element
  *
  * @param args
  */
 public static void main(String[] args)
 {
  int[] input =
  { 2, 5, 6, 1, 3 };
  int length = input.length;
  ArrayList < ArrayList < Integer > > table =
   new ArrayList < ArrayList < Integer > >();

  for (int i = 0; i < length; ++i)
  {
   ArrayList < ArrayList < Integer > > tempTable =
    new ArrayList < ArrayList < Integer > >();
   for (ArrayList < Integer > j : table)
   {
    if (j.get(j.size() - 1) <= input[i])
    {
     ArrayList < integer > temp = new ArrayList < integer >();
     temp.addAll(j);
     temp.add(input[i]);
     tempTable.add(temp);
    }
   }
   table.addAll(tempTable);
   ArrayList < Integer > temp =
     new ArrayList < Integer >();
   temp.add(input[i]);
   table.add(temp);
  }

  // output
  for (ArrayList < Integer > i : table)
  {
   for (Integer j : i)
   {
    System.out.print(j + ", ");
   }
   System.out.println();
  }
```

```
 }

}
```

Jumping frog problem

Problem

There were n stones across a river. Each stone was places 1 meter apart. A frog can jump in steps of 1 meter, 2 meters, 3 meters and so on. The first jump is always 1 meter. The typical behavior of the frog is such that if it goes k meters in one jump then on its next jump it can go either k-1, k or k+1 meters. It won't jump backward. So when there were n stones across the river it could cross the river. Let's take n=10; and initial position of the frog is 0.
Some possible paths could be
1, 2, 3, 4, 6, 9, 10
1, 3, 6, 10
1, 2, 4, 7, 10
Now some of the stones across the river have been removed. Find the algorithm to figure out whether the frog would be able to cross the river after removing those stones. For example, as the frog always takes the first jump for 1 meter. If the first stone is removed then it will never be able to cross the river.Solution
Solution description

Solution

If the frog somehow reaches an intermediate stone m and his last jump was for k meters. Then his next location can be m+k-1, m+k or m+k+1 (Jump can't be 0 or negative, so those cases will be ignored). So we can create a subproblem of reaching from m to the end. As we start with initial position 0 and initial jump 1, for different branches the same value of m and k can appear multiple times. As we have overlapping subproblems we can use dynamic problem. We can use a two dimensional array to store result for different m and k values for memoization of the repeated subproblems.

Code

```
import java.util.ArrayList;
import java.util.List;

public class FrogJump {

 public static void main(String[] args) {
  boolean[] stone = { true, false, true, true, true, false, true, false,
    true, false, true, true, false, true };
  System.out.println("Frog jump " + frogJump(stone));
 }

 static List frogJump(boolean[] stone) {
  int[][] jumpTable = new int[stone.length + 1][stone.length + 1];
  if (frogJump(stone, 1, 0, jumpTable)) {
   List result = new ArrayList();
   int currentStone = 0;
   int jump = 1;
   while (currentStone < stone.length) {
    result.add(currentStone);
    jump = jumpTable[currentStone][jump];
    currentStone += jump;
   }
   result.add(stone.length);
   return result;
  }
  return null;
 }

 static boolean frogJump(boolean[] stone, int jump, int currentLocation,
   int[][] jumpTable) {
  if (currentLocation >= stone.length
    || jumpTable[currentLocation][jump] != 0)
   return true;
  if (jump < 1 || !stone[currentLocation])
   return false;
```

```
  for (int i = 1; i > -2; --i) {
   if (frogJump(stone, jump + i, currentLocation + jump + i, jumpTable)) {
    jumpTable[currentLocation][jump] = jump + i;
    return true;
   }
  }
  return false;
 }
}
```

Distributed doubly linked list sum

Problem

You are given a doubly linked list whose nodes are distributed. Every node has next, previous pointers and a method send(integer). A node can talk to its next and previous nodes only. Different instances of same threads are running in them. How would you implement the run method of the thread class so that each node prints the sum of complete linked list.

Solution

We will accumulate the number from right to left and then propagate the sum from left to right.

1. Every node except the right most node will wait for a first send call. Right node will send its value immediately to its left node.
2. After receiving the number from right side each node will add it with its own value and send it to its left.
3. The left most node, after receiving the send call, add its own number and send the complete sum to its right node.
4. All nodes will wait for a second send call.
5. After receiving the second send call each node will print the sum.
6. After printing each node will propagate the sum to its right node.

The threads will start running as soon as the nodes are created. But the doubly linked list may not be completely formed by then. So we will use a shared flag which will signal all the threads to start working.

Code

```
public class DistributedDoublyLinkedListSum
{
 public static volatile boolean startFlag = false;

 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  a.next = b;
  b.next = c;
  c.next = d;
  d.next = e;
  e.prev = d;
  d.prev = c;
  c.prev = b;
  b.prev = a;
  startFlag = true;
 }

 private static class Node
 {
  Node next;
  Node prev;
  int value;
  Integer data;
```

```
 public Node(int value)
 {
  this.value = value;
  new Thread(new NodeRunner(this, value)).start();
 }

 public synchronized void send(int data)
 {
  this.data = data;
 }

}

private static class NodeRunner implements Runnable
{
 Node node;
 int id;

 public NodeRunner(Node node, int id)
 {
  this.node = node;
  this.id = id;
 }

 @Override
 public void run()
 {
  while (!startFlag)
  {
  }
  if (node.next == null)
  {
   node.prev.send(node.value);
  } else
  {
   while (node.data == null)
   {
    try
    {
     Thread.sleep(10);
    } catch (InterruptedException e)
    {
    }
   }
   int sum = node.data + node.value;
   if (node.prev != null)
   {
    node.data = null;
    node.prev.send(sum);
   }
  }
  while (node.data == null)
  {
   try
   {
    Thread.sleep(10);
   } catch (InterruptedException e)
   {
   }
  }
  System.out.println("id:" + id + " sum:" + node.data);
  if (node.next != null)
  {
   node.next.send(node.data);
  }

 }
}
```

```
}
```

Distributed binary tree sum

Problem

You are given a binary tree where each node has an integer value, a left, right and parent pointer. Every node is an independent distributed system where a thread is running in each node. You can talk to other node only by one method called "send(node, data)". And a node can call "send" only to its children or parent. How will you design the system so that all the nodes know the total sum of values of all the nodes in the binary tree and report them asynchronously.

Solution

The basic technique that we will follow to get the total sum is as follows.
1. Every node will wait for its children to report the sum of nodes of the tree rooted at the child node.
2. The node will add the sum of its left and right children and add its own value and send it to its parent.
3. Then wait for its parent to report back the total sum of the tree.
The root node will be the first to know the complete sum. Then it propagates it back to its children which in turn will propagate down to its children till every node is aware of the total sum.

To implement this, we keep on adding the values coming via the send method and increment a counter to record how many times the send has been called. When the send has been called equal to the number of children, it means all children has reported their sums. So a node will send the sum after adding its own value to its parent (So leaf nodes will immediately send its own value to its parents). Then resets the sum. Then it keeps on waiting for another send call from its parent. When the send method is called again, the sum will be equal to the total sum of the tree. Then it propagates the same value by calling its children and print the value to console. We need to start the process after our tree building is completed. So we maintained a global static flag which signals all threads to start their operations.

Code

```
public class DistributedSystemSum
{
 public static volatile boolean startFlag = false;

 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  Node f = new Node(6);
  Node g = new Node(7);
  Node h = new Node(8);
  a.left = b;
  b.parent = a;
  a.right = c;
  c.parent = a;
  b.left = d;
  d.parent = b;
  c.left = e;
  e.parent = c;
  c.right = f;
  f.parent = c;
  f.left = g;
  g.parent = f;
  f.right = h;
  h.parent = f;
  startFlag = true;
 }
```

```
private static class Node
{
 Node parent;
 Node left;
 Node right;
 int value;
 int sum = 0;
 int receiveCount = 0;

 public Node(int value)
 {
  this.value = value;
  NodeRunner nodeRunner = new NodeRunner(this, value);
  new Thread(nodeRunner).start();
 }

 public synchronized void send(Integer data)
 {
  sum += data;
  receiveCount++;
 }

 public synchronized int getReceivedCount()
 {
  return receiveCount;
 }
}

private static class NodeRunner implements Runnable
{
 Node node;
 int id;

 NodeRunner(Node node, int id)
 {
  this.node = node;
  this.id = id;
 }

 @Override
 public void run()
 {
  while (!DistributedSystemSum.startFlag)
  {
  }
  int childCount = 0;
  if (node.left != null)
   childCount++;
  if (node.right != null)
   childCount++;
  while (node.getReceivedCount() != childCount)
  {
  }
  int sum = node.sum + node.value;
  node.sum = 0;
  if (node.parent != null)
  {
   node.parent.send(sum);
   while (node.getReceivedCount() != childCount + 1)
   {
   }
  } else
  {
   node.sum = sum;
  }
  if (node.left != null)
   node.left.send(node.sum);
  if (node.right != null)
```

```
    node.right.send(node.sum);
   System.out.println("Thread id=" + id + " sum=" + node.sum);

  }

 }
}
```

Distributed Circular linked list sum

Problem

You are given a circular linked list whose nodes are distributed. Every node has next pointer and a method send(integer). A node can talk to its next node only. Different instances of same threads are running in the nodes. How would you implement the run method of the thread class so that each node prints the sum of complete linked list.

Solution

We will accumulate the number along the next pointer circularly and then propagate it in the same direction in second iteration.

The trick of this problem is that in a circular linked list every node is equal and there is no starting and ending node. For this, we have taken a flag object, Every thread will try to acquire its lock, and whoever wins in this race will become a decisive node and it changes the value of the flag so that no other thread can become decisive. So there will be only one node randomly chosen in the linked list which will start the operation. It will send its value to the next node. All other nodes will wait for its first send call. After that it adds its own value and send it to the next node. In this way the decisive node will be the first to know the complete sum. It will then send it to the next node. So again every node will wait for second send call. Then it will print the total sum and propagate it to its next node and exits.

Code

```
public class DistributedCircularListSum
{
 public static volatile boolean startFlag = false;
 private static Boolean flag = false;

 public static void main(String[] args)
 {
  Node a = new Node(1);
  Node b = new Node(2);
  Node c = new Node(3);
  Node d = new Node(4);
  Node e = new Node(5);
  a.next = b;
  b.next = c;
  c.next = d;
  d.next = e;
  e.next = a;
  startFlag = true;
 }

 private static class Node
 {
  Node next;
  int value;
  Integer data;

  public Node(int value)
  {
   this.value = value;
   new Thread(new NodeRunner(this, value)).start();
  }
```

```
  public synchronized void send(int data)
  {
   this.data = data;
  }

 }

 private static class NodeRunner implements Runnable
 {
  Node node;
  int id;

  public NodeRunner(Node node, int id)
  {
   this.node = node;
   this.id = id;
  }

  @Override
  public void run()
  {
   while (!startFlag)
   {
   }

   boolean isFirst = false;
   synchronized (flag)
   {
    if (flag == false)
    {
     flag = true;
     isFirst = true;
    }
   }
   if (isFirst)
    node.next.send(node.value);
   else
   {
    while (node.data == null)
    {
     try
     {
      Thread.sleep(10);
     } catch (InterruptedException e)
     {

     }
    }
    int sum = node.value + node.data;
    node.data = null;
    node.next.send(sum);
   }
   while (node.data == null)
   {
    try
    {
     Thread.sleep(10);
    } catch (InterruptedException e)
    {

    }
   }
   System.out.println("id:" + id + "sum:" + node.data);
   node.next.send(node.data);
  }
 }
}
```