Question 1: Graphs

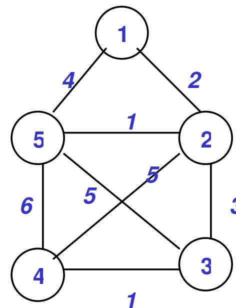
The edge weight list below describes an undirected weighted graph, i.e. all connectios have to be considered bi-directional. Create the correxponding distance matrix by hand and draw the graph. An edge weight list differs ffrom an edge list in that it contains triplets node, node, weight instead of node pairs.

A well-known algorithm for computing the shortest distance between all nodes works as follows: For every node in the graph, try to use it as intermediate node between all pairs of nodes. If the distance via the intermediate node is shorter than the distance recorded in the matrix, replace the matrix entry. Write a static method which transforms a distance matrix given as parameter into a path matrix recording shortest paths using this algorithm.

What is the complexity of the algorithm?

Edge weight list: [5, 8; 1, 2, 2; 1, 5, 4; 2, 3, 3; 3, 4, 1; 4 ,5, 6; 2, 5, 1; 2, 4, 5; 3, 5, 5]

	1	**2**	**3**	**4**	**5**
1	max	2	max	max	4
2	2	max	3	5	1
3	max	3	max	1	5
4	max	5	max	max	6
5	4	1	5	6	max




```
/* paramter is distance matrix,
 * node numbers starting with 1,
 * will be transformed into matrix of minimal distances
 */
public static void floyd (int[][]distances) {
      for (int mid=1; mid<distances.length; mid ++)
            for (int start=1; start<distances.length; start++)
                  for (int end=1; end<distances.length; end++)
                        distances[start][end] =
                              Math.min(distances[start][end],
                                    distances[start][mid]+ distances[mid][end];
}
```

Complexity is $O(n^3)$ – 3 nested loops.

Question 2: Backtacking

Here is "OpFlips", a simpified "MatchMaths" problem: A formula consists of n integer operands combined by + or – operators, and an integer result, such as "17-3-4+2 = 14". The task is to find all correct fomulas that can be formed by flipping two operators from + to – or vice versa, e.g. "17+3-4-2=14".

Given the following class Formula, write a recursive backtracking method

```
static void printSolutions(Formula form, int position, int flipsDone).
```

which prints out all correct tranformations of the formula **and comment on its complexity**.The starting call would be `printSolutions(form, 0, 0);` also write the method `evaluate` in `Formula.`

```
class Formula {
      public  int numops;       // number of operators in the formula
      private boolean [] ops;   // length numops, array of operators, true is +
      private int[] numbers;    // length numops+2, all operands and the result

      public Formula(String input, int numops) {
      // creates numbers and ops and stores the input formula there
      }
```

```
        public void flipOp(int pos) { ops[pos] = !ops[pos]; // flip it
        }
        // effect can be undone by calling flipOp(pos) again;

        public void evaluate() {
        // prints formula, if mathematically correct
        }
}
```

Hint: Undoing the flips saves copying the formula for every recursive call.

Extra question – 5 extra points: implement evaluate().

```
static void printSolutions(Formula form, int position, int flipsDone) {
      if (flipsDone==2) form.evaluate();        // no more flips possible
      if (position==form.numops-1) form.evaluate();   // end of formula
      printSolutions(form, position+1, flipsDone);    // recurse unflipped
      form.flipOp(position);                          // flip current op
      printSolutions(form, position+1, flipsDone-1);  // recurse flipped
      form.flipOp(position);                          // undo flip;
}
```

Backtracking has always exponential complexity $O(k^n)$, k being the number of choices per search step, and n the max. depth of each search path. It may be optimized by dynamic programming.

```
public void evaluate() {
      int termValue = numbers[0];
      String formula = ""+numbers[0];
      for (int argindex=1; argindex<numbers.length-1; argindex++)
      {     if (ops[argindex-1]) {
                  termValue += numbers[argindex];
                  formula += " + "+numbers[argindex];
            }
            else {
                  termValue -= numbers[argindex];
                  formula += " - "+numbers[argindex];
            }
      }
      if (termValue == numbers[numbers.length-1]) { // math.correct
            System.out.println
                  (formula + " = " + numbers[numbers.length-1]);
      }
}
```

Question 3: Linked Structures

1. ***Briefly*** describe the advantages of using a double linked list over a single linked list. What is the function of an anchor node?

2. Assume a folowing class SortedList :

```
class SortedList {
   private class Node {
       String info;
       Node succ;
       Node pred;
       Node(String info)
       {      this.info = info;    }
   }

   protected static final String UNDEFINED = "###-Undefined-###";
   protected Node anchor = new Node(UNDEFINED);
```

```
        public void add(String info)
        { /* adds a Node containing info */ }
        }
        public boolean isEmpty() { /* true if the tree has no nodes */ }

    }
```

Implement the two methods in bold type, add, and isEmpty. **What is the complexity of the add method?**

Hint: String implements `public int compareTo(String otherString)`*. It yields –1 if the current String is lexicographically smaller, +1 if it is larger, and 0 if the Strings are equal.*

```
        public void add(String info) {
        Node newnode = new Node(info);
        Node pred = getPredecessorRec(info, anchor);
//      Node pred = getPredecessorIt(info);   // iterative alternative
        newnode.pred = pred;
        newnode.succ = pred.succ;
        pred.succ = newnode;
        newnode.succ.pred = newnode;
}

private Node getPredecessorRec(String info, Node sublist) {
        if (sublist.succ == null) return sublist;
        if (! (sublist.succ.info.compareTo(info) > 1)) return sublist;
        return getPredecessorRec(info, sublist.succ);
}

private Node getPredecessorIt(String info) {
        Node result = anchor;
        while (result.succ != null) {
              if (!(result.succ.info.compareTo(info) < 1)) return result;
              result = result.succ;
        }
        return result;
}
public boolean isEmpty() { /* true if the tree has no nodes */ }
        return anchor.succ = null;
}
```

The complexity of the add method is linear, O(n), because the list has to be linearly traversed to find the insertion point.

Question 4: Clever Data Structures

The class below is an IndexedTable, i.e. it contains of an unsorted list of (bulky) entries, and an index table of the same size which contains a sorted order for the bulky list entries. The idea is to do sorting by moving lists of indeces rather than lists of bulky entries. Write the add method which adds an entry, keeping the index table sorted.

```
class IndexedTable {

    private String[] content = new  String[size];
    private int [] index = new int[size];

    public void add(String entry)
    { /* add entry to content
          keeping the index table sorted by insertion
      */
     }
}
```

What is the complexity of finding an entry?

Extra question – 5 extra points: Use binary search to find the insertion point in the index table.

```
class IndexedTable {
    private int size = 100;
    private int contentIndex = 0;
    private String[] content = new  String[size];
    private int [] index = new int[size];

    /* add entry to content
     * keeping the index table sorted by insertion
     */
    public void add(String newEntry)  {
      if (contentIndex == size) throw new RuntimeException("Table Full");
      content[contentIndex] = newEntry;     // enter content
      int newIndex = findSuccessor(newEntry);
      for (int i=newIndex; i<size-1; i++)
            index[i+1] = index[i];          // move
      index[newIndex] = contentIndex;       // enter index
      contentIndex ++;
     }

    /* binsearch */
    private int findSuccessor(String entry) {
      if (contentIndex == 0) return 0;
      return findSuccRec(entry, 0, contentIndex-1);
    }

    private int findSuccRec(String entry, int bottom, int top) {
      int mid = bottom + (top-bottom)/2;
      int compare = entry.compareTo(content[mid]);
      if (mid == top)                // only one element
            return compare < 0 ? top : top+1;
      if (compare == 0) return mid;  // identical is succ
      if (compare > 0)
            return findSuccRec(entry, mid+1, top);
      if (compare < 0)
            return findSuccRec(entry, bottom, mid);
      return mid;                    // formally required
    }

    /* linear search */
    private int findSuccessorIt(String entry) {
      if (entry.compareTo(content[0]) <= 0) return 0;  // entry is smallest
      for (int index=1; index<contentIndex; index++)
         if (entry.compareTo(content[index]) <= 0) return index;
      return contentIndex;            // entry is largest
    }
```

Depending on the search algorithm, the complexity of adding an entry is either O(n) for linear search or O(log(n) for binsearch.

Question 5: Theory

Answer **5 of these 6** quesitons by marking true or fals and giving a **brief explanation**. No points are given if the explanation is missing.

True ()
False(x

1. The advantage of true random numbers is that they are reproducable

Explanation *pseudo random numbers are reproducable, true random numbers useue stochastic input and are not reproducable.*

True ()
False(x)

2. Dynamic programming reduces the computation time by distributing the computation over several. processors.

Explanation *Distributed prgramming uses several processes, dynamic programming stores intermediate results*

True ()
False(x)

3. An RPN formula consists of a list of operands followed by a list of operators.

Explanation *In an RPN formula, each operator follows its argument subterms*

True (x)
False()

4. Tree balancing is only relevant on sorted trees.

Explanation *In an unsorted tree, entries can be evenly distributed over all leaves to maintain balance.*

True (x)
False()

5. Logarithmic complexity is usually achieved by using the divide and conquer pattern.

Explanation *Log complexity comes from reducing the complexity recursively by a factor, usually 2, by dividing the problem into subproblems which can be combined easily, e.g with linear complexity.*

True ()
False(x)

6. JUnit testing is a good alternative to debugging.

Explanation *Testing is about showing a program's correctness, debugging about removing errors*