CS 608 Algorithms and Computing Theory

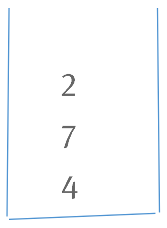
Week 4: Stacks

1. Stacks

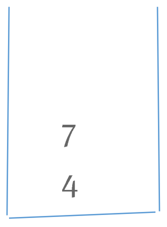
A stack is a special kind of list in which items can be inserted from one end, and the only item that can be retrieved is the last item inserted.

A stack is like a cylinder with one end open and the other closed. You can insert items from the open end. When retrieved the item which was inserted last comes out.

Suppose the numbers 4, 7, and 2 are inserted into a stack:



4 goes in the bottom, 7 goes on top of 4, 2 goes on top of 7. When an item is retrieved, 2 comes out. The stack looks like this after retrieving 2:



2. Operations on a stack

Common stack operations are:

Operation	Explanation
push(x)	Inserts x on to the stack
pop()	Retrieves an element from the stack (the last element which went in)

empty()	Returns true if the stack is empty; returns false otherwise
topElement()	Returns the top element of the stack without removing the element from the stack

We will implement a stack using an array and implement these operations as methods. A stack is an ADT with the property last in first out (LIFO).

3. Implementation of a stack

We will implement a stack using an array. You can also implement using a linked list.

This implements a stack as a class, **myStack**, of integer element. We have also included exception handling and a **main**() method to test the code.

Fields:
Two fields.

private int[] *elements*;
This is the array which keeps the elements of the stack. We will decide the size of the array in constructor (see below).

private int *top*;
This variable points to the top element of the stack. It is just the subscript of the top element in the array.

Constructor:
One constructor:

```
myStack(int size) {
        if (size <= 0)
            throw new IllegalArgumentException( "Stack's capacity must be positive");
        elements = new int[size];
        top = -1;
}
```

This constructs a stack of specified size. If the size ≤ 0, it gives an error.

Methods:

We will write methods for the operations, push(x), pop(), empty() and top().

```
void push(int value) {
        if (top == elements.length)
            throw new StackException("Stack is full");
        top++;
        elements[top] = value;
}



int pop() {
       if (top == -1)  throw new StackException("Stack is empty");
       return(elements[top--]);
}

boolean isEmpty() {
        return (top == -1);
}

int topElement() {
        if (top == -1)  throw new StackException("Stack is empty");
        return elements[top];
}

public class StackException extends RuntimeException {
        public StackException(String message) {
            super(message);
        }
}

//Now the main method
public static void main(String[] args){
        System.out.println("MyStack");
        myStack stack1 = new myStack(10);
        stack1.push(15);
        stack1.push(20);
        stack1.push(13);
        stack1.push(30);
        stack1.push(10);
        System.out.println("top element top element: "+stack1.topElement());
        System.out.println("Top element retrived pop(): "+stack1.pop());
        System.out.println("Top element retrived pop(): "+stack1.pop());
        System.out.println("Top element retrived pop(): "+stack1.pop());
        System.out.println("Top element retrived pop(): "+stack1.pop());
        System.out.println("top element top element: "+stack1.topElement());
    }
}
```

Output:

```
MyStack
top element top element: 10
Top element retrived pop(): 10
Top element retrived pop(): 30
Top element retrived pop(): 13
Top element retrived pop(): 20
top element top element: 15
```

As you see the implementation of a stack using an array is very simple. In spite of its by simplicity, stacks are used extensively (mainly by systems software).

We will see three important applications of the ADT stacks: Conversion of infix expressions into postfix, evaluations of postfix expressions and evaluation of infix expressions.

4. Postfix expressions

The traditional algebraic expressions we write are in the form a + b, where a and b are operands and + is an operator. In this form, the operator appears between the operands. This form expressions are called infix expressions.

There is another kind of representing algebraic expressions, called postfix expressions. In postfix expressions, operator comes after the two operands:

The infix expression a + b is expressed in postfix as follows: a b +. Postfix expressions are also called Reverse Polish notation (RPN).

Before we discuss algorithm to convert infix to postfix, let us see how we can do it manually.

Manually converting infix to postfix

Manually converting an infix expression to postfix involves two steps:

1. First completely parenthesis the infix expression. This means include every operation in a pair of parenthesis keeping in mind the precedence rules of the operators.

   Examples:

   a + b * c = (a + (b * c))

   a + b * c + d = (((a + (b * c)) + d)

   Notice that the parentheses dictate the order of operations; there is no ambiguity. Once the expression is completely parenthesized, there is also no need to remember any precedence rules.

2. Second step is, move all the operators to the corresponding right parenthesis position and remove all parentheses.

   Example:

   Take the completely parenthesized expression, (((a + (b * c)) + d).

   Let me spread it to make the diagram clearer:

$$\begin{tikzcd}[column sep=tiny]
(& (& (& a & + \arrow[rrrrrr, bend right] & (& b & * \arrow[rr, bend right] & c &) &) & + \arrow[rr, bend right] & d &)
\end{tikzcd}$$

Move the operators as shown and drop all the parentheses:

a b c * + d +

This is the postfix equivalent to a + b * c + d

Evaluating postfix expressions

Before we discuss algorithm to evaluate postfix expressions, let us see how we can do it manually.

Consider a postfix expression,

a b c * + d +

Assume values a = 2, b = 4, c = 3 and d = 1.

Scan the expression left to right until hit an operator, then perform the operation on the two previous operands, remove the two operands and the operator and leave the answer right there. And continue scanning and performing operation.

Example:

Scan left to right

a b c $\underset{\Uparrow}{*}$ + d +

We hit the operator shown by the arrow. Perform b * c, that is 12. Keep the answer right there and continue scanning.

a 12 $\underset{\Uparrow}{+}$ d +

We encounter a +. Perform a + 12, that is 14. Keep the answer right there and continue scanning,

14 d $\underset{\Uparrow}{+}$

We encounter a +. Perform 14 + d, that is 15, Keep the answer right there and continue scanning. But, it is the end of the expression. We stop. The final answer is left there.

15.

Practice Examples:

Manually convert the following infix expressions to postfix:

1. a+b*(c+d)
2. a+b*(c+d*(a+d))
3. a+b/c+d
4. (a+b)/(c+d)

Manually evaluate the following postfix expressions

Assume a=2, b=3, c=2, and d=4.

1. a b + c d + a + *
2. a b a c + * c d + * +
3. a b c d + * +
4. a b + c d b * + *

5. An algorithm to convert an infix expression to postfix

Let assume the following:

Operands are one letter characters, like a, b, c, and d. Operators permitted are +, -, * and /. The usual operator precedence rules apply (As we all know, the operations + and - have the same precedence. The operations * and / have equal precedence. Each of * and / is has higher precedence than each of + and -.)

We will use character stack.

We scan input expression left to right, one character at a time, and store the character scanned in a variable ***currentCharacter***.

1. If the **currentCharacter** is an operand, output it. Continue scanning.
2. If the **currentCharacter** is an operator do one of the following:
    a. If the stack is empty, push the **currentCharacter** on to the stack. Continue scanning.
    b. If the currentCharacter has higher precedence than the topElement of the stack, push the **currentCharacter** on to the stack. Continue scanning.
    c. If the **currentCharacter** has lower than or equal to the precedence of the topElement of the stack, pop the stack and output the element. Repeat this

until either #2a or #2b become true. Push the **currentCharacter** on to the stack. Continue scanning.

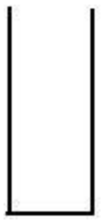
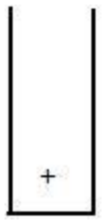
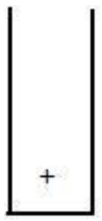
3. Continue scanning and steps 1, and 2, until you reach the end of the input string. Stop scanning and pop the stack and output the element until the stack becomes empty.

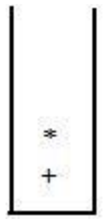
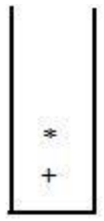
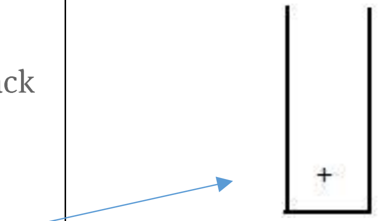
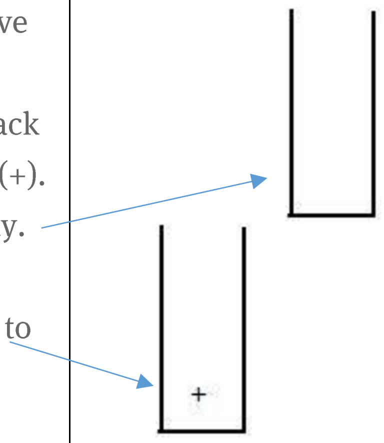
Example:

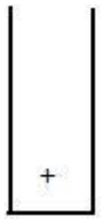
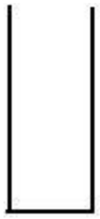
Given infix expression: a + b * c + d

Let us follow the algorithm step by step:

We scan left to right, one character at a time.

currentCharacter	**What to do?**	**Output and stack status**
currentCharacter = a.	Case #1. Output **a.** Continue scanning.	Output: **a** Stack: Empty 
currentCharacter = +.	Stack is empty. Case #1a: Push + on to the stack. Continue scanning.	Output: **a** Stack: 
currentCharacter = b	Case #1. Output **b.** Continue scanning.	Output: **a b** Stack: 

currentCharacter = *	topElement: + * Has higher precedence than + Case #2b: Push * on to the stack. Continue scanning.	Output: **a b** Stack: 
currentCharacter = c	Case #1. Output **c.** Continue scanning.	Output: **a b c** Stack: 
currentCharacter = +	topElement = * + Has lower precedence than * Case #2c: Pop the stack and output element. Check again, topElement = + topElement and currentCharacter have equal precedence. Case #2c: Pop the stack and output element (+). Stack becomes empty. Push the currentCharacter on to the stack. Continue scanning.	Output: **a b c** * Stack:  Output: **a b c** * + Stack: 

currentCharacter = d	Case #1. Output **d.** End of input string reached.	Output: **a b c** * + **d** Stack 
End of the input string is reached	Case #3. Pop stack and output until stack becomes empty.	Output: **a b c** * + **d** + Stack 

Postfix expression: **a b c** * + **d** +

Program implementing the algorithm to convert infix to postfix

The following program implements the algorithm step by step so that it is easy to understand.

```
public class infixToPostfix {
        public static void main(String[] args) {
                myStack operatorsStack = new myStack(10);
                String[] infixExpressions =
                        { "a+b*c+d", "a+b*c+d*e", "a*b+c+d*e", "a*b*c/d+e", "a/b*c/d*e+f" };
                String infixExpression;
                for (int k = 0; k < infixExpressions.length; k++) {
                        infixExpression = infixExpressions[k];
                        int inputLength = infixExpression.length();
                        String output = "";
                                for (int i = 0; i < inputLength; i++) {
                                        char currentCharacter = infixExpression.charAt(i);
```

```
                    if (currentCharacter >= 'a' && currentCharacter <= 'z')
                    output += currentCharacter;
                    else {
                    switch (currentCharacter) {
                    case '+':
                    case '-':
                        if (operatorsStack.isEmpty()) {
                            operatorsStack.push(currentCharacter);
                        } else {
                            while (!operatorsStack.isEmpty())
                            output += (char) operatorsStack.pop();
                            operatorsStack.push(currentCharacter);
                        }
                    break;
                    case '*':
                    case '/':
                if (operatorsStack.isEmpty() || operatorsStack.topElement() == '+'
                            || operatorsStack.topElement() == '-')
                        operatorsStack.push(currentCharacter);
                    else {
            while (!operatorsStack.isEmpty() && operatorsStack.topElement() != '+'
                                && operatorsStack.topElement() != '-')
                            output += (char) operatorsStack.pop();
                        operatorsStack.push(currentCharacter);
                    }
                    break;
                    }
                    }
            }
            while (!operatorsStack.isEmpty())
                output += (char) operatorsStack.pop();
            System.out.println("Given Infix expression: " + infixExpression);
            System.out.println(" Postfix Expression: " + output);
```

```
        }
    }
}

class myStack {
    private int top;
    private char[] operatorsArray;

    public myStack(int size) {
        operatorsArray = new char[size];
        top = -1;

    }

    public void push(char op) {
        top++;
        operatorsArray[top] = op;
    }

    public char pop() {
        return operatorsArray[top--];
    }

    int topElement() {
        return operatorsArray[top];
    }

    public boolean isEmpty() {
        return (top == -1);
    }
}
```

The program tests for several input infix expressions.

Output:

```
Given Infix expression: a+b*c+d
 Postfix Expression: abc*+d+
Given Infix expression: a+b*c+d*e
 Postfix Expression: abc*+de*+
Given Infix expression: a*b+c+d*e
 Postfix Expression: ab*c+de*+
Given Infix expression: a*b*c/d+e
 Postfix Expression: ab*c*d/e+
Given Infix expression: a/b*c/d*e+f
 Postfix Expression: ab/c*d/e*f+
```

Note: Observe that I have written myStack as a separate class. You can place it in a separate file (myStack.java) and leave it in the same package as this program.
Note: The program works on 5 expressions.

6. An algorithm to evaluate a postfix expression

Evaluating a postfix expression is very simple. Just scan left to right, when you encounter an operator perform the operation on the two previous operands. Continue until the entire expression is canned and you are left with an answer.
There is no need to think about operation precedence. This is because the postfix expression implicitly contains precedence. Actually, we took care of operator precedence when we converted an infix expression into postfix.

Let us assume operands are one letter characters, like a, b, c, and d. Operators permitted are +, -, * and /. We are also given the values of the operands.

We will use on stack of the type **float.**
We scan the given postfix expression left to right one character at a time. Let us denote the character being scanned, ***currentCharacter***.

If the currentCharacter is an operand push the value of the operand on to the stack.
If the currentCharacter is an operator, pop the stack twice and perform the operation, and push the result back to the stack. The operation is performed as follows:

pop2 operator pop1, where pop1 is the first operand popped and pop2 is the second.

Let us implement this algorithm in Java:

```
public class postfixEvaluation {
        public static void main(String[] args)  {
                myNewStack valuesStack = new myNewStack(10);
          String[] postfixExpressions ={
                  "abc*+d+", "ab*c*d+", "abc*d*+", "abc/d*+","ab/c*d/"
          };
          float a = 2.5f, b = 1.5f, c=5f, d=3f;
          float value=0;
          float result = 0;
          float left = 0, right = 0;
          String postfixExpression;
          for(int k=0;k<postfixExpressions.length; k++){
              value = 0;
              result = 0;
              left = 0;
              right = 0;
              postfixExpression = postfixExpressions[k];
          int inputLength = postfixExpression.length();
          for (int i = 0; i < inputLength; i++) {
            char currentCharacter = postfixExpression.charAt(i);
            if (currentCharacter =='a') {valuesStack.push(a);continue;}
            if (currentCharacter =='b') {valuesStack.push(b);continue;}
            if (currentCharacter =='c') {valuesStack.push(c);continue;}
            if (currentCharacter =='d') {valuesStack.push(d);continue;}

            if(currentCharacter !='a' &&
                    currentCharacter !='b' &&
                    currentCharacter !='c' &&
                    currentCharacter !='d')
```

```
                     {
                    right = valuesStack.pop();
                    left = valuesStack.pop();
                    switch(currentCharacter){
                    case '+': result = left + right; break;
                    case '-': result = left - right; break;
                    case '*': result = left * right; break;
                    case '/': result = left / right; break;
                    }
                    valuesStack.push(result);
                    }
                }
                 System.out.println("Given postfix expression: "+postfixExpression);
                 System.out.println("Values of a, b, c, and d: "+a+" "+" "+b+" "+c+" "+d);
                 System.out.println("Final Answer: "+ valuesStack.pop());
                }
            }
}

class myNewStack {
            private int top;
  private float[] valuesArray;

  public myNewStack(int size) {
            valuesArray = new float[size];
    top = -1;

  }
  public void push(float val) {
    top++;
    valuesArray[top] = val;
  }
```

```
 public float pop() {
   return valuesArray[top--];
 }
 float topElement() {
    return valuesArray[top];
}
 public boolean isEmpty() {
   return (top == -1);
 }
 public void makeStackEmpty(){
        top = -1;
 }
 }
```

Output:

```
Given postfix expression: abc*+d+
Values of a, b, c, and d: 2.5  1.5 5.0 3.0
Final Answer: 13.0
Given postfix expression: ab*c*d+
Values of a, b, c, and d: 2.5  1.5 5.0 3.0
Final Answer: 21.75
Given postfix expression: abc*d*+
Values of a, b, c, and d: 2.5  1.5 5.0 3.0
Final Answer: 25.0
Given postfix expression: abc/d*+
Values of a, b, c, and d: 2.5  1.5 5.0 3.0
Final Answer: 3.4
Given postfix expression: ab/c*d/
Values of a, b, c, and d: 2.5  1.5 5.0 3.0
Final Answer: 2.7777777
```

Note: Observe that I have written myNewStack as a separate class. You can place it in a separate file (myNewStack.java) and leave it in the same package as this program.
Note: The program works on 5 expressions.

7. An algorithm to evaluate infix expressions

Let us discuss an algorithm to evaluate a given infix expression directly without converting it into a postfix expression first.

Preliminaries:

Our assumptions: The infix expression contains one letter operands, permitted operators are +, -, *, /, and ^ (exponentiation), with the usual precedence rules, parentheses are permitted. We use two stacks, *opndstk* for operands and *oprtrstk* for operators.

Let us use a, b, c, and d for operands. Values for a, b, c, and d are given as **double**. The *opndstk* is a stack of **double** and oprtrstk is of type **char**.

Algorithm to evaluate a given infix expression:

Step 1 – Scan the input infix expression left to right, one character at a time, until the end of the infix expression is reached. Let us denote the character being scanned the *currentToken*. Perform only one of the cases (a) through (f) on the *currentToken*:

(a) If the *currentToken* is an operand, push it onto the *opndstk* stack.
(b) If the *currentToken* is an operator, and the *oprtrstk* stack is empty then push it onto the *oprtrstk* stack.
(c) If the *currentToken* is an operator and the *oprtrstk* stack is not empty, and the *currentToken's* precedence is greater than the precedence of the stack top of *oprtrstk*, then push the *currentToken* on to *oprtrstk*.
(d) If the *currentToken* is '(', then push it on to *oprtrstk*.
(e) If the *currentToken* is '), then process as explained below in **Step 2** until the a '(' is encountered in *oprtrstk*. AT this stage **pop()** the *oprtrstk*, and ignore '('.
(f) If cases (a), (b), (c), (d), and (e) do not apply, then process as explained in **Step 2**.

Step 2 - Process: **pop()** *opndstk* twice, call the elements opr2 and opd1 respectively. **Pop()** oprtrstk once, call the element obtained *oprtr*. Perform opd1 *oprtr* opd2, and push() the result onto *opndstk*.

Step 3: At the end of the infix string, keep repeating Step 2 until the oprtrstk stack becomes empty. The element final left in the *opndstk* is the result of the given expression.

Implementing this algorithm in Java is left as an assignment.

8. Java built-in Stack class

Java provides a built-in Stack class, in the **java.util** package. The Stack class defines the usual stack operations: **push(*element*), pop(), peek(), and empty().** The **peek**() operator is what we called **topElement**(). For details, see Oracle manual: https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html

As an example, let us rewrite **postfixEvaluation** program, using built-in Java Stack class.

```
package stacksPackage;
import java.util.*;
public class evalPostfixUsingJavaStack {

       public static void main(String[] args)  {
            Stack valuesStack = new Stack();
           String[] postfixExpressions ={
                 "abbc+*+", "ab*c*d+", "abc*d*+", "abc/d*+","ab/c*d/"
           };
           float a = 3f, b = 4f, c=2f, d=3f;
           float value=0;
           float result = 0;
           float left = 0, right = 0;
           String postfixExpression;
           for(int k=0;k<postfixExpressions.length; k++){
            value = 0;
            result = 0;
            left = 0;
            right = 0;
            postfixExpression = postfixExpressions[k];

           int inputLength = postfixExpression.length();

           for (int i = 0; i < inputLength; i++) {
               char currentCharacter = postfixExpression.charAt(i);
               if (currentCharacter =='a') {valuesStack.push(a);continue;}
               if (currentCharacter =='b') {valuesStack.push(b);continue;}
               if (currentCharacter =='c') {valuesStack.push(c);continue;}
               if (currentCharacter =='d') {valuesStack.push(d);continue;}

               if(currentCharacter !='a' &&
                      currentCharacter !='b' &&
                    currentCharacter !='c' &&
                    currentCharacter !='d')
                  {
                right = (Float)valuesStack.pop();
                left = (Float) valuesStack.pop();
                switch(currentCharacter){
                case '+': result = left + right; break;
                case '-': result = left - right; break;
                case '*': result = left * right; break;
                case '/': result = left / right; break;
                }
                valuesStack.push(result);
                }
          }
           System.out.println("Given postfix expression: "+postfixExpression);
           System.out.println("Values of a, b, c, and d: "+a+" "+" "+b+" "+c+"
"+d);
           System.out.println("Final Answer: "+ valuesStack.pop());
          }
      }
}
```

Note:

These are only differences between **postfixEvaluation program** and the following **evalPostfixUsingJavaStack** program:

1. We don't use our implementation of Stack class. Instead we use Java built-in Stack class. This is how we create a Stack class:

   Stack *valuesStack* = **new Stack**();

2. The elements of Stack are objects. So when we retrieve values, we need to convert them appropriately. We have used:

   right = (Float)valuesStack.pop();

   left = (Float) valuesStack.pop();

9. Exercises

1. Manually convert each of the following infix expressions to postfix

   (a) (a+b)*(a+b*c)

   (b) (a*b)*(a*b+c)

   (c) (a – b) (c / d*a)

   (d) (a + b) / c*d + e

   (e) a + b / (c + d)+a

2. Manually evaluate each of the following postfix expressions for given values of variable
   (a) a b + a b * c + * for a = 1, b = 2, and c = 3
   (b) a b + c / d * a + for a = 0, b = 2, and c = 1
   (c) a – b c d / a * for a = 1, b = 3, c = 1, and d = 2
   (d) a b c d + / + a + for a = 1, b = -1, c = 2, and d = -2
   (e) a b + c / d e + * for a = 1, b = 2, c = 3 and d = 1

3. Find the value of the unknown variable:
   (a) The value of the postfix expression abc*+d+ is 10 and b=1, c=5 and d=3, find the value of a.
   (b) The value of the postfix expression ab+c*d* is 48 and a=1, b=3 and d=3, find the value of c.
   (c) The value of the postfix expression ab+abc*+* is 12 and a=2, and b=1 find the value of c.
   (d) The value of the postfix expression ababc*+*+ is 32 and b=4, and c=2find the value of a.
   (e) The value of the postfix expression abab+c**+ is 19 and a=1, and b=2 find the value of c .

4. This question relates to our algorithm converting infix to postfix
   What do you do if,
   currentCharacter is the operand b and topElement on the stack is a +
   Pick the right answer:
   (a) Push the currentCharacter on to the stack
   (b) Send the currentCharacter to output
   (c) Pop stack once and output, and then push the currentCharacter to the stack
   (d) None of the above

5. This question relates to our algorithm converting infix to postfix
   What do you do if,
   currentCharacter is + and topElement on the stack is a *

Pick the right answer:

(a) Push the currentCharacter on to the stack
(b) Send the currentCharacter to output
(c) Pop stack once and output, and then push the currentCharacter to the stack
(d) None of the above

6. This question relates to our algorithm converting infix to postfix
   What do you do if,
   currentCharacter is * and topElement on the stack is a +

   Pick the right answer:

   (a) Push the currentCharacter on to the stack
   (b) Send the currentCharacter to output
   (c) Pop stack once and output, and then push the currentCharacter to the stack
   (d) None of the above

7. This question relates to our algorithm converting infix to postfix
   What do you do if,
   currentCharacter is * and topElement on the stack is a *

   Pick the right answer:

   (a) Push the currentCharacter on to the stack
   (b) Send the currentCharacter to output
   (c) Pop stack once and output, and then push the currentCharacter to the stack
   (d) None of the above

8. **Stack sortable permutation**: Here is a popular among mathematicians/computer scientists. We will present a simple version of the problem. Given an input sequence of numbers 1 2 3, how many combinations can you output using one stack and any sequence of push() and pop() operations.

   For example:
   Given 1 2 3
   Perform the following sequence of push() and pop() operations:
   push(element)

push(element)
pop()
pop()
push()
pop()
This result in the output sequence: 3 1 2

The given number 1 2 3 has 6 permutations: 1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 2 1 and 3 2 1.
(If there are n elements, the number of permutations is n!)
Now the question: Given 1 2 3, which permutation cannot be output?

9. Same as problem 8, with four numbers: 1 2 3 4.

You can skip problems 8 and 9.