- To see the java codes and their test IO, click here .

Problem 1. [35%]

a) Both algorithms are shown below (in a more compact form). Inductively follow the chain of pre/post-conditions of each line shown in red.

```
Algorithm turnOff(n)
// Pre-Cond:   Lights 1..n are on.
// Post-Cond:  Lights 1..n are off.
    if n <= 0 then return
                                  // 1111…1111111
    turnOff( n – 2 )              // 0000…0000011
    Turn off light n              // 0000…0000010
    turnOn( n – 2 )               // 1111…1111110
    trunOff( n – 1 )              // 0000…0000000
end
```

```
Algorithm turnOn(n)
// Pre-Cond:   Lights 1..n are off.
// Post-Cond:  Lights 1..n are on.
    if n <= 0 then return
                                  // 0000…0000000
    trunOn( n – 1 )               // 1111…1111110
    turnOff( n – 2 )              // 0000…0000010
    Turn on light n               // 0000…0000011
    turnOn( n – 2 )               // 1111…1111111
end
```

b) The Java code appears in **Lights.java**. The sample outputs appear in **LightsTestIO.txt**.

c) **Analysis**: As noted in the forum, by "analysis" we always mean "mathematical analysis on all input instances", not just experimental demo of a few cases and then a jump to some general conclusion without further justification.

We apply the technique of LS5, pages 27-30. We first set up recurrence relations that express the number of times lights are switched by each algorithm. Then we derive the closed form solution and estimate the asymptotic growth rate.

Part 1: *set up the recurrence:*

Let $T_{off}(n)$ and $T_{on}(n)$ denote the number of times lights are switched by the respective algorithms. We set up their recurrences according to the recursive structure of each algorithm:

$$T_{off}(n) = \begin{cases} 0 & if\ n \le 0 \\ T_{off}(n-2) + 1 + T_{on}(n-2) + T_{off}(n-1) & if\ n > 0 \end{cases}$$

$$T_{on}(n) = \begin{cases} 0 & if\ n \le 0 \\ T_{on}(n-1) + T_{off}(n-2) + 1 + T_{on}(n-2) & if\ n > 0 \end{cases}$$

Both recurrences become identical if we simply drop the subscripts "off" and "on". This shows $T_{off}(n) = T_{on}(n)$, for all n. So, we simply call both of them $T(n)$. The simplified recurrence is:

$$T(n) = \begin{cases} 0 & if\ n \le 0 \\ 2T(n-2) + T(n-1) + 1 & if\ n > 0 \end{cases}$$

Examples:

$$\begin{aligned} T(-1) &= T(0) = 0, \\ T(1) &= 2T(-1) + T(0) + 1 = 0 + 0 + 1 = 1, \\ T(2) &= 2T(0) + T(1) + 1 = 0 + 1 + 1 = 2, \\ T(3) &= 2T(1) + T(2) + 1 = 2 + 2 + 1 = 5, \\ T(4) &= 2T(2) + T(3) + 1 = 4 + 5 + 1 = 10. \\ T(5) &= 2T(3) + T(4) + 1 = 10 + 10 + 1 = 21. \end{aligned}$$ ■

c) **Part 2:** *solve the recurrence by the **guess-&-verify** method of LS5, pages 27-30:*

Similar to the Fibonacci recurrence, it shouldn't be hard to conjecture that $T(n)$ grows exponentially in n. So, we **guess** the general solution form $T(n) = c + r^n$, for some constants c and r to be determined. (The purpose of adding constant c is to turn the inhomogeneous recurrence into a homogeneous one as we shall see below.) We now **verify** the guess by plugging it in the general recurrence

$$T(n) = 2T(n-2) + T(n-1) + 1 .$$

We get

$$c + r^n = 2(c + r^{n-2}) + (c + r^{n-1}) + 1 \quad \Longrightarrow \quad r^n - r^{n-1} - 2r^{n-2} = 2c + 1$$

With $c = -1/2$, the right hand side simplifies to 0 (homogenized), which simplifies the left hand side to:

$$r^2 - r - 2 = 0.$$

This quadratic has two roots: $r_1 = 2$, $r_2 = -1$.

By homogeneity, the general solution of the recurrence for $T(n)$ has the form

$$T(n) = a\, r_1{}^n + b\, r_2{}^n + c, \qquad c = -1/2 .$$

To determine the constants a and b, plugin-&-solve the two boundary conditions $T(-1) = T(0) = 0$. We get $a = 2/3$, $b = -1/6$. Hence, the exact solution is:

$$T(n) = \frac{2}{3}2^n - \frac{1}{6}(-1)^n - \frac{1}{2} = \left\lfloor \frac{2^{n+1}}{3} \right\rfloor .$$

We conclude: $T(n) = \Theta(2^n)$.

d) The above two algorithms are combined and converted into one:

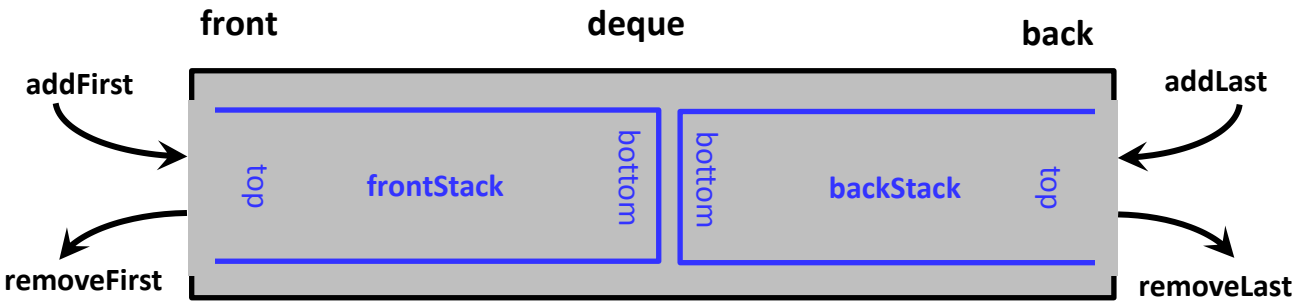
```
Algorithm flipSwitches(n, s)
    // boolean state s means "on" if true, "off" if false.
    // Pre-Cond:  Lights 1 .. n  are in state  not s.
    // Post-Cond: Lights 1..n  are in state  s.
    if n <= 0 then return
    if s then flipSwitches(n – 1, true)
    flipSwitches(n – 2, false )
    Flip the switch of light n          // i.e., L[n] ← 1 – L[n]
    flipSwitches(n – 2, true)
    if not s then flipSwitches(n – 1, false)
end
```

Problem 2. [30%]

Remark: **deque** (pronounced "deck", not "dequeue") stands for "double-ended-queue". We can add or remove elements from both ends of a deque. We can implement a deque efficiently in many ways, e.g., by circular arrays or doubly linked lists as explained in [GTG §6.3]. However, in this assignment we are asked to implement a deque by a pair of stacks.

Design Idea: The idea, as depicted in the figure below, is to maintain the following **invariant:**

The sequence of elements from front to back of the deque (from left to right in the figure below) corresponds to the sequence of elements from top to bottom of frontStack, followed by the sequence of elements from bottom to top of backStack.



Methods addFirst & addLast are performed by a push on frontStack & backStack, respectively. We perform removeFirst by a pop on frontStack if it's not empty. If frontStack is empty but backStack is not, we first transfer **all** elements of backStack into frontStack by a pop-push sequence. The effect of this **"double-reversal"** transfer is that it maintains the invariant (the "left-to-right" sequence of elements in the picture). Now we can pop from frontStack. If both stacks are empty, that indicates the deque is empty. The operation removeLast can be handeled symmetrically.

Time Analysis: With this implementation, all deque operations listed on page 248 of [GTG] can be done in O(1) time, except first, removeFirst, last and removeLast. These methods in the worst case take O(n) time, where n is the current number of elements in the deque. This running time is due to the fact that all elements reside on the "other" stack and with a pop-push sequence they should first be transferred to the desired stack.

Java Code: **NewDeque.java** is the source code, and
NewDequeTestIO.txt shows some test results .

Problem 3. [35%]

Design Idea: The idea behind RPIEtoFPIE is as follows. We first push the input RPIE into stack inStk starting with a dummy operator character '$'. Then we start to pop and process individual characters, named by variable c in the code. They come out of inStk in reverse order. After character c is processed, it will be pushed onto stack processedStk along with appropriately added left parentheses. Generally speaking, operators and right parentheses are pushed into stack symbolStk first. This helps to determine where to place the left parentheses as follows. If character c is an operator, then we repeatedly pop pairs of matching operator followed by right parentheses from symbolStk, and push a matching left parentheses onto processedStk. We continue this (inner-loop) iteration until the top of symbolStk is not an operator. We then push the operator c on top of symbolStk.

Ignoring parentheses, the expression must alternate between operand and operator, beginning and ending with an operand. Also, right parentheses should not immediately follow an operator in the input expression. We use the boolean operandLastProcessed to check these syntactic conditions.

The outer-loop terminates when inStk is empty. Then we remove the dummy operator '$' that appears at the top of both stacks symbolStk and processedStk. The input is a valid RPIE iff symbolStk is now empty. The contents of the output FPIE are in processedStk but in reverse order. Once again, we pop the content of this stack and return its string version.
(Note: no syntax checking is done on operand character sequences.)

The idea behind RPIEtoUPPE is even simpler. As we read the input RPIE (left-to-right), we append the operands on the output and stack up its operators into opStk. Each time we encounter a right parentheses in RPIE, we pop the latest operator from opStk and append it to the output.

Time Analysis: Each of the three methods, main, RPIEtoFPIE, RPIEtoUPPE takes O(n) time, where n is the length of the input string as a RPIE. The reason is that we spend O(1) time per input character. To see this, notice that (even in the presence of double nested loops in RPIEtoFPIE) each method uses a constant number of stacks, and each input character is pushed into and popped from these stacks at most once. These stack operations dominate the entire computation time. Hence, the total running time is O(n).

Important Note: *Remember that String is immutable. If in your algorithm you do repeated String cut-&-paste operations, they will generate new strings with running time proportional to the total length of the strings involved in the operation. If such cut-&-paste ops are repeated many times over strings of growing length, the total running time may become* ***quadratic or worse****, not* ***linear****.* ■

Java Code: **Expression.java** is the source code, and
ExpressionTestIO.txt shows some test results .