CS29003 Algorithms Laboratory

Warm-Up Assignment

For Test Submission on 09–January–2018

You push $1, 2, \ldots, n$ (in that order) to an initially empty stack S. You also pop n times from S in between and after the push operations. You never make an attempt to pop in a situation when S is empty. Just before every pop, you print the current top of the stack. During the n push and the n pop operations, you print a permutation of $1, 2, \ldots, n$. Let us call such a permutation *stacky*. Not all permutations of $1, 2, \ldots, n$ are stacky. For example, for $n = 4$, the permutation $2, 4, 3, 1$ is stacky, whereas the permutation $2, 4, 1, 3$ is not. The initial goal of this assignment is to count and print stacky permutations of a given length n.

One possibility is to generate all possible permutations of $1, 2, \ldots, n$, and test each generated permutation for stackiness (a stackiness test is to be developed in Part 4). Instead of following this strategy, we adopt an alternative and somewhat more efficient approach.

Part 1: For an allowed sequence of n push and n pop operations, generate a string T of length $2n$ as follows. T is initially empty. For each push operation, append **+** to T, and for each pop operation, append **–** to T. At the end, T contains exactly n plus symbols and exactly n minus symbols. Moreover, since pop is never allowed from an empty stack, to the left of every position in T, the number of plus symbols is $\geqslant$ the number of minus symbols. Let us again call such a string T a *stacky string of parameter n*. It is evident that stacky permutations of $1, 2, \ldots, n$ are in one-to-one correspondence with stacky strings of parameter n (and length $2n$). Therefore if we can count stacky strings, the count of stacky permutations is also available.

Write a recursive function **`stackystr`** to generate and print all stacky strings of a given parameter n, and also to return their count. You start with an empty string T, and then recursively keep on adding new symbols at the end of T. Appending of **+** is allowed if and only if the current number of plus symbols in T is $< n$, whereas appending of **–** is allowed if and only if T currently contains more plus symbols than minus symbols. For each allowed appending, make a recursive call on the appended T. At depth $2n$ of recursion, you have added n plus and n symbols to T. Print T. Also increment a count (must not be global or static) to remember the generation of a stacky string.

Part 2: If you are interested only in the count $C(n)$ of stacky strings of parameter n (which is equal to the count of stacky permutations of length n), you do not need to generate the stacky strings at all. It is well-known that $C(n)$ is equal to the n-th *Catalan number* given by the formula $C(n) = \dfrac{1}{n+1}\binom{2n}{n}$. Write an efficient function **`stackystrformula`** that, upon the input of n, returns the count $C(n)$ computed using this formula.

Part 3: Now we come to our original goal of printing all stacky permutations of length n. Copy the function **`stackystr`** to a function called **`stackyperm`**. Replace the printing of T at recursion depth $2n$ by printing of the stacky permutation that corresponds to the generated string T. Use a stack S, and guided by the **+** and **–** symbols in T, make appropriate push and print-and-pop operations.

Part 4: Write an efficient function **`isstacky`** that, upon the input of an integer array A of length n storing a permutation of $1, 2, \ldots, n$, decides whether the input permutation is stacky or not. Notice that unlike in Part 3, here the input is a permutation of $1, 2, \ldots, n$, and not a string of **+** and **–** symbols.

The *main*() function

- Read a positive integer $n \leqslant 10$ from the user.
- Call **`stackystr`** to print all stacky strings of parameter n. Also print the count returned by the outermost call of the function.
- Make three individual calls of **`stackystrformula`** with inputs n, 16, and 32, and print the three return values.
- Call **`stackyperm`** to print all stacky permutations of length n. Also print the count returned by the outermost call of the function.

- Read a few permutations of $1,2,\ldots,n$ from the user (or generate them randomly), and invoke **`isstacky`** to obtain and print the decision whether each permutation is stacky or not.

Sample output

```
n = 4

+++ All stacky strings of parameter 4 are:
    Stacky string     1: +-+-+-+-
    Stacky string     2: +-+-++--
    Stacky string     3: +-++--+-
    Stacky string     4: +-++-+--
    Stacky string     5: +-+++---
    Stacky string     6: ++--+-+-
    Stacky string     7: ++--++--
    Stacky string     8: ++-+--+-
    Stacky string     9: ++-+-+--
    Stacky string    10: ++-++---
    Stacky string    11: +++---+-
    Stacky string    12: +++--+--
    Stacky string    13: +++-+---
    Stacky string    14: ++++----
--- There are 14 of them

+++ Stacky string counts by formula
    C[4] = 14
    C[16] = 35357670
    C[32] = 555340648770481 98

+++ All stacky permutations of length 4 are:
    Stacky permutation     1: 1,2,3,4
    Stacky permutation     2: 1,2,4,3
    Stacky permutation     3: 1,3,2,4
    Stacky permutation     4: 1,3,4,2
    Stacky permutation     5: 1,4,3,2
    Stacky permutation     6: 2,1,3,4
    Stacky permutation     7: 2,1,4,3
    Stacky permutation     8: 2,3,1,4
    Stacky permutation     9: 2,3,4,1
    Stacky permutation    10: 2,4,3,1
    Stacky permutation    11: 3,2,1,4
    Stacky permutation    12: 3,2,4,1
    Stacky permutation    13: 3,4,2,1
    Stacky permutation    14: 4,3,2,1
--- There are 14 of them

+++ Check for stacky permutations:
    Permutation 3 2 4 1 is stacky
    Permutation 4 1 2 3 is not stacky
```

Submit a single C/C++ source file. Do not use global/static variables.