CS29003 Algorithms Laboratory

Assignment No: 2

Date: 21–January–2019

Recursive Formulation of Algorithms

This exercise deals with a problem that can be formulated recursively. You may translate that formulation to either a recursive function or an iterative function.

Let n be a positive integer. We want to find out in how many ways n can be written as an arithmetic expression involving the only operand 1 and two binary operators: addition and multiplication. For example, here are all the fundamentally different ways of realizing $n = 10$ using such arithmetic expressions.

$$\begin{aligned}
10 &= 1+1+1+1+1+1+1+1+1+1 \\
&= 1+1+1+1+1+1+(1+1)\times(1+1) \\
&= 1+1+1+1+(1+1)\times(1+1+1) \\
&= 1+1+(1+1)\times(1+1+1+1) \\
&= 1+1+(1+1)\times(1+1)+(1+1)\times(1+1) \\
&= 1+1+(1+1)\times(1+1)\times(1+1) \\
&= 1+(1+1+1)\times(1+1+1) \\
&= (1+1)\times(1+1+1+1+1) \\
&= (1+1)\times(1+(1+1)\times(1+1)) \\
&= (1+1)\times(1+1)+(1+1)\times(1+1+1)
\end{aligned}$$

We may allow certain permutations of the summands. For example, $(1+1)\times(1+1)+1+1+(1+1)\times(1+1)$ may be called a representation different from $1+1+(1+1)\times(1+1)+(1+1)\times(1+1)$. However, multiplication by 1 is not allowed. For example, $(1+1)\times(1+1+1+1+1)\times 1$ is not treated as a valid expression. Moreover, the addition operator does not require parentheses, so removal of unnecessary parentheses may covert two expressions to the same representation. For example, $(1+1)+(1+1)\times(1+1+1+1)$, $1+1+(1+1)\times(1+(1+1)+1)$, and $1+(1+(1+1)\times((1+1)+(1+1)))$ all simplify to the same expression $1+1+(1+1)\times(1+1+1+1)$.

The objective of this assignment is to generate all representations of a given positive integer n, and eventually identify one expression having the smallest number of 1's.

Part 1: Decide a data structure to store all representations of a given n. Each expression can be stored in a character array (that is, as a string). You need to use string library functions for handling these strings.

Part 2: Write a function *findallexpr*(n) to generate all valid arithmetic expressions of n. This function is to be based on a recursive formulation. In order to generate all expressions for n, we assume that all expressions for i are available for $i = 1, 2, 3, \ldots, n-1$. Then, we take all i, j in the range $1 \leqslant i \leqslant j \leqslant n-1$ with $i + j = n$. Then, an expression for n can be obtained by concatenating an expression for i followed by $+$ followed by an expression for j. Likewise, if n is composite, we write $n = ij$ for $2 \leqslant i \leqslant j < n$. We then concatenate an expression for i, the multiplication sign, and finally an expression for j to get an expression for n. Since multiplication has higher precedence than addition, the expressions for i and j should be parenthesized if needed. For example, all parentheses in $(1+1)\times(1+(1+1)\times(1+1))$ are needed, whereas $1+1+(1+1)\times(1+1)\times(1+1)$ does not require parenthesization like $1+1+(1+1)\times((1+1)\times(1+1))$.

This strategy generates many duplicate expressions. For example, for $n = 10$, we have two additive representations $10 = 1+9$ and $10 = 2+8$. The expressions $1 = 1$, $2 = 1+1$, $8 = 1+1+1+1+1+1+1+1$, and $9 = 1+1+1+1+1+1+1+1+1$ generate the same expression $1+1+1+1+1+1+1+1+1+1$ for 10. The same representation can also be generated by $10 = 3+7 = 4+6 = 5+5$. Therefore whenever an expression is generated, check whether it is not already generated earlier. If not, add the new expression to the list of representations of n. Some summand/factor permutations cannot be avoided by this strategy. For example, $10 = 4+6 = 4+(2+4) = (1+1)\times(1+1)+1+1+(1+1)\times(1+1)$ and $10 = 2+8 = 2+(4+4) = 1+1+(1+1)\times(1+1)+(1+1)\times(1+1)$ are essentially the same expression but not identical as strings, and so are generated as two different expressions. Removing this type of duplicates calls for quite some programming effort which may be avoided in this assignment.

The function *findallexpr*(n) should iteratively generate all expressions for $1,2,3,\ldots,n$. It would eventually return only all the expressions for n.

Part 3: Write a function *printallexpr* to print all expressions in the sequence provided.

Part 4: Our basic objective is to identify an expression for n with the minimum possible number of 1's. To that effect, sort the list of expressions for n in the descending order of numbers of 1's in the expressions. If two expressions have the same number of 1's, then the longer of these two expressions should be placed earlier. After this sorting, the last expression has the smallest number of 1's, and is the shortest among all expressions with the same number of 1's. You may use any sorting algorithm. But whatever algorithm you choose, you must implement it. Do not use any built-in sorting routine like *qsort*.

The *main*() function

- Read n from the user.
- Call *findallexpr* with parameter n.
- Print the expressions in the order they are generated.
- Sort the list of expressions by your function of part 4.
- Print the sorted list of expressions.

Sample output

```
n = 10

+++ Before sorting
10  = 1+1+1+1+1+1+1+1+1+1
    = 1+1+1+1+1+1+(1+1)*(1+1)
    = 1+1+1+1+(1+1)*(1+1+1)
    = 1+1+(1+1)*(1+1)+1+1+1+1
    = 1+1+(1+1)*(1+1)+(1+1)*(1+1)
    = 1+1+(1+1)*(1+1+1+1)
    = 1+1+(1+1)*(1+1)*(1+1)
    = 1+(1+1)*(1+1)+1+1+1+1+1
    = 1+(1+1)*(1+1)+1+(1+1)*(1+1)
    = 1+(1+1+1)*(1+1+1)
    = (1+1)*(1+1)+1+1+1+1+1+1
    = (1+1)*(1+1)+1+1+(1+1)*(1+1)
    = (1+1)*(1+1)+(1+1)*(1+1+1)
    = (1+1)*(1+1+1+1+1)
    = (1+1)*(1+(1+1)*(1+1))
... 15 expressions

+++ After sorting
10  = 1+1+(1+1)*(1+1)+(1+1)*(1+1)
    = 1+(1+1)*(1+1)+1+(1+1)*(1+1)
    = (1+1)*(1+1)+1+1+(1+1)*(1+1)
    = 1+1+1+1+1+1+(1+1)*(1+1)
    = 1+1+(1+1)*(1+1)+1+1+1+1
    = 1+(1+1)*(1+1)+1+1+1+1+1
    = (1+1)*(1+1)+1+1+1+1+1+1
    = 1+1+1+1+1+1+1+1+1+1
    = (1+1)*(1+1)+(1+1)*(1+1+1)
    = 1+1+1+1+(1+1)*(1+1+1)
    = 1+1+(1+1)*(1+1)*(1+1)
    = 1+1+(1+1)*(1+1+1+1)
    = (1+1)*(1+(1+1)*(1+1))
    = 1+(1+1+1)*(1+1+1)
    = (1+1)*(1+1+1+1+1)
... 15 expressions
```

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