DESIGN AND ANALYSIS OF ALGORITHMS — HT 2019
Practical: A String Bracketing Problem

Let $\Sigma = \{A, B, C\}$. The elements of Σ have the following multiplication table:

		Right-hand symbol		
		A	B	C
Left-hand symbol	A	B	B	A
	B	C	B	A
	C	A	C	C

Thus $AB = B$, $BA = C$, and so on. Note that the multiplication defined by this table is neither commutative nor associative.

Now we want to consider the following questions:

- Given a word $w = x_1 x_2 \cdots x_n$ of symbols of Σ ($w \in \Sigma^*$) and a symbol $z \in \Sigma$, is it possible to parenthesise w in such a way that the value of the resulting expression is z?
- Given a word $w = x_1 x_2 \cdots x_n \in \Sigma^*$ and a symbol $z \in \Sigma$, in how many different ways (possibly none) is it possible to parenthesise w such that the value of the resulting expression is z?
- If it is possible to parenthesise a given word $w = x_1 x_2 \cdots x_n \in \Sigma^*$ in such a way that the value of the resulting expression is $z \in \Sigma$ determine one such possible bracketing.

We will use the terms parenthesising and bracketing interchangeably.

As an example, if $w = ABBA$, then there are five ways of bracketing which produce the following values:

$$\begin{aligned} (A(B(BA))) &= B \\ (A((BB)A)) &= A \\ ((AB)(BA)) &= A \\ ((A(BB))A) &= C \\ (((AB)B)A) &= C \end{aligned}$$

There are two ways of bracketing to get an A and two to get a C, whereas there is only one way to get a B.

As another example, the five ways of bracketing $BBBB$ all yield B, so it is not possible to get an A or a C from $BBBB$.

The purpose of this practical is to design and implement efficient algorithms in Scala to answer these bracketing questions. In particular, your program should eventually output information like this:

```
Bracketing values for ABBA
A can be achieved in 2 ways
B can be achieved in 1 way
C can be achieved in 2 ways
```

Getting started

The file `Brack.scala` can be found on the course website:
`https://www.cs.ox.ac.uk/teaching/materials18-19/algdesign/`

Create a directory `daa` in your home directory, change into that directory, and copy the file `Brack.scala` to your directory from the course website.

You can compile the Scala file using

```
> fsc Brack.scala
```

and can execute the resulting program using

```
> scala Brack -command [file]
```

The program expects a command flag for which portion of the code to run. This is so you can seperately test the different implementations asked for in the practical. In addition the program can take an optional argument of a file name for specific test cases. If a file name is not given, the program takes standard input from the command line.

Initially, the program parses a string from either standard IO or the text file and turns it into an array of characters called `plain`. For convenience in the other procedures, `plain` is turned in to an array of Integers `plainInt` where A, B, C correspond to $0, 1, 2$ respectively (it halts if it encounters other characters in the input string).

For clarity in the procedure definitions, we define the array `op` to contain our multiplication rules. In addition we also let `MAXWORD` be the maximal length of strings input to the program.

Task 1

Write a recursive procedure

```
def PossibleRec(w: Array[Int], i: Int, j: Int, z:Int): Boolean
```

such that `PossibleRec`(w, i, j, z) returns `TRUE` if there exists a bracketing of $w[i..j)$ that gives the result z, and returns `FALSE` if no such bracketing exists.

This is not yet the efficient version of the procedure, but you should check your procedure with some well-chosen examples.

Hint: For a word of length one, only the symbol itself can be produced. For a word w of length greater than one, w can be parenthesised to produce z if there exists a splitting of w into two parts, $u = w[i..k)$ and $v = w[k..j)$, such that

- u can be parenthesised to produce x, and
- v can be parenthesised to produce y, and
- the product of x and y is z.

So we have to consider all possible splittings of w into two parts and all possible ways to get z as a binary product.

You can execute `PossibleRec` on standard input using

```
> scala Brack -PossibleRec
> BACB
  Bracketing values for BACB
  A is not possible
  B is possible
  C is possible
```

Task 2

Write a recursive procedure

```
def NumberRec(w: Array[Int], i: Int, j: Int, z:Int): Int
```

such that `NumberRec`(w, i, j, z) returns the number of distinct ways of parenthesising $w[i..j)$ to obtain z.

You can execute `NumberRec` on standard input using

```
> scala Brack -NumberRec
> BACB
  Bracketing values for BACB
  A can be achieved in 0 ways
  B can be achieved in 1 way
```

```
C can be achieved in 4 ways
```

Task 3

In terms of the length n of the word w, how much time should your procedures `PossibleRec`$(w, 0, n, z)$ and `NumberRec`$(w, 0, n, z)$ take, and why? Verify your prediction by running your program with suitable tests.

Hint: To time the execution of the program with file input `testcase` (containing $ABBA$ in this case), use the shell command `time`:

```
> time scala Brack -NumberRec testcase
Bracketing values for ABBA
A can be achieved in 2 ways
B can be achieved in 1 way
C can be achieved in 2 ways

real 0m0.376s
user 0m0.421s
sys  0m0.060s
```

The exact format of the timing information will vary depending on the shell you are using. We are interested in the 'user time' (0.421s in the example above), which is approximately the amount of CPU time used by our program itself. The figure may vary a bit if you run the program several times, but it should provide a reasonable estimate of the running time.

Task 4

In order to design a more efficient algorithm, we have to avoid the recalculation of intermediate results. We define an array

```
var poss = Array.ofDim[Boolean](MAXWORD, MAXWORD, 3)
```

for this table of results.

Write a dynamic programming procedure

```
def Tabulate(w: Array[Int], n: Int): Unit
```

to fill in this table such that for all $0 \leq i < j \leq n$ and for all $z \in \{0, 1, 2\}$, the entry `poss`$(i)(j)(z)$ indicates whether or not there is a bracketing of $w[i..j)$ evaluating to z.

The current `Tabulate` skeleton code is written for Task 5 and beyond where you extend your dynamic programming algorithm to compute the ways of bracketing (and optionally print out an example bracketing). To test `Tabulate` for the array `poss` feel free to change the skeleton code.

You can execute `Tabulate` using

```
> scala Brack -Tabulate
```

Task 5

Modify your dynamic programming algorithm to also use a table

```
var ways = Array.ofDim[Int](MAXWORD, MAXWORD, 3)
```

such that for all $0 \leq i < j \leq n$ and for all $z \in \{0, 1, 2\}$, the entry $\texttt{ways}(i)(j)(z)$ stores the number of distinct ways of bracketing $w[i..j)$ to obtain z.

You can execute `Tabulate` using

```
> scala Brack -Tabulate
> BACB
  Bracketing values for BACB
  A can be achieved in 0 ways
  B can be achieved in 1 way
  C can be achieved in 4 ways
```

Task 6

What is the maximum word length for which your program can correctly determine the number of bracketings?

How much time should your new dynamic programming algorithm take in terms of the length n of the input w? Verify your prediction with suitable tests.

How does the dynamic programming version compare to the recursive version? Discuss your findings.

Task 7 (optional)

Enhance your dynamic programming algorithm to produce an example of a bracketing whenever one exists, *e.g.*

```
> scala Brack -Tabulate testcase
Bracketing values for ABBABBAABA
A can be achieved 1906 ways
For example:(A(B(B((AB)(B((A(AB))A))))))
B can be achieved 1197 ways
For example:(A(B(B(A(B(B((A(AB))A)))))))
C can be achieved 1759 ways
For example:((AB)(B((AB)(B((A(AB))A)))))
```

Your algorithm can return a different bracketing example, if there is more than one possibility.

Hint: You may want to use the array

```
var exp = Array.ofDim[BinaryTree](MAXWORD, MAXWORD, 3)
```

to store the example expressions as binary trees.

Report

Your report should consist of your well commented file `Brack.scala`, some interesting test results, and answers to the questions in Task 3 and Task 6.

Acknowledgments

Thanks to Michael Vanden Boom, Mike Spivey, Ani Calinescu, Tom Melham, Barney Stratford, and Bill McColl for designing and improving this practical over the years, and to Francisco Marmolejo Cossío for preparing the Scala version of this practical.

Giulio Chribella, HT 2019