IOI’98 – Setubal/Portugal

Problem Portfolio

Polygon

1. Introduction to the Problem

POLYGON aims at testing the technical skills in Algorithms. More precisely, it is intended to check if the contestants understand the essentials of Complexity Theory and if they are able to apply some of the classical algorithm design techniques (namely, divide and conquer, memory function, or dynamic programming).
It is not difficult to realise that this problem deals with arithmetic expressions. Basically, the first move gives rise to an expression without brackets (which is, in general, ambiguous) and the subsequent moves correspond to the evaluation of a fully parenthesised version of that expression. Since the number of distinct first moves in a polygon with N vertices is N (thus, polynomially related to the size of the input), the crucial point is to find out, among the set of the fully parenthesised versions of each of those expressions, which is the highest represented value.

POLYGON has been classified by the Scientific Committee in the following way.

Problem type:	Algorithmic
Problem understanding	Difficult
Algorithm effort	Difficult
Implementation effort	Medium
Number of possible solutions per test	1
Several levels of testing	Yes

The next issue concerns algorithms for solving POLYGON. Let then N be the number of vertices of the input polygon, and E be an arithmetic expression without brackets, generated after the first move. Obviously, E comprises N integers and N-1 operators. As we have already mentioned, the question is how to compute the maximum value of the fully parenthesised versions of E (referred to as parenthesizations of E).

2. Algorithms

2.1. Greedy description

Due to the presence of positive and negative integers, none of the obvious greedy strategies seem to work. Consequently, even if there are algorithms of this sort, they have not been studied.

2.2. Brute force

A **naive** (yet, the most difficult to implement) algorithm generates and evaluates all the possible parenthesizations of the expression. Needless to say, it runs in exponential time, given that the number of distinct parenthesizations of an arithmetic expression with k operators is the Th Catalan number (and it is well known that the Catalan numbers grow exponentially).
There is an implementation of such an algorithm in Pascal.

According to the **divide and conquer** approach, we may try to compute the maximum value of E, recursively, interms of the maximum values of the sub-expressions of E. In this case, although additions do not raise any problem, in that

$$\mathrm{Max}(E_1+E_2) = \mathrm{Max}(E_1)+\mathrm{Max}(E_2),$$

the rule for products is not so simple, because negative numbers are allowed. Nevertheless, the following equalities hold:

$$\mathrm{Max}(E_1*E_2)=\max\{\mathrm{Max}(E_1)*\mathrm{Max}(E_2), \mathrm{Max}(E_1)*\mathrm{Min}(E_2),\mathrm{Min}(E_1)*\mathrm{Max}(E_2),\mathrm{Min}(E_1)*\mathrm{Min}(E_2)\},$$
$$\mathrm{Min}(E_1*E_2)=\min\{\mathrm{Max}(E_1)*\mathrm{Max}(E_2), \mathrm{Max}(E_1)*\mathrm{Min}(E_2), \mathrm{Min}(E_1)*\mathrm{Max}(E_2), \mathrm{Min}(E_1)*\mathrm{Min}(E_2)\}.$$

Besides,

$$\mathrm{Min}(E_1+E_2) = \mathrm{Min}(E_1)+\mathrm{Min}(E_2),$$

and, if I denotes an integer (that is to say, an expression without operators),

$$\mathrm{Max}(I) = \mathrm{Min}(I) = I.$$

Moreover, since every expression $E' = I_1\ o_1\ I_2\ o_2\ I_3 \cdots I_k\ o_k\ I_{k+1}$, with k operators, can be seen in k different ways,

$$(F_1)\ o_1\ (F_2\ o_2\ F_3 \cdots F_k\ o_k F_{k+1}),$$
$$(F_1\ o_1\ F_2)\ o_2\ (F_3 \cdots F_k\ o_k\ F_{k+1}),$$
$$\cdots,$$
$$(F_1\ o_1\ F_2 o_2\ F_3 \cdots F_k)\ o_k\ (F_{k+1}),$$

we must compute the maximum and the minimum values of each one of them, in order determine the maximum and the minimum values of E'. Notice that, apart from the specific arithmetic calculations, this problem is very similar to the Matrix-Chain Multiplication one.
It is not difficult to verify that a direct implementation of this algorithm generates many equal recursive calls. As a result, it also falls in the exponential pit (like, for instance, the direct recursive implementation of the Fibonnacci function).
This algorithm has been implemented both in C and in Pascal.

2.3. *Polynomial Algorithms —O(N^4)*

To overcome the drawbacks caused by equal recursive calls, we can apply two distinct techniques: **memory function** or **dynamic programming**. The resulting algorithms run in O(N^3) time and require O(N^2) space, as expected. All the same, the time complexity of POLYGON turns out to be O(N^4), because N problems (one for each generated expression) are solved.
The two solutions have been implemented both in C and in Pascal.

2.4 *Polynomial Algorithms —O(N^3)*

At first sight, we can be lead to believe that expressions generated by different first moves are independent. However, after a close examination, we conclude that they actually have many common sub-expressions. So, we can calculate the maximum and the minimum values of each sub-expression only when they are needed for the first time, and use them thereafter, saving a lot of computations. In practice, as the number of distinct sub-expressions is N^2, both the memory function and the dynamic programming algorithms for solving POLYGON run in $O(N^3)$ time and require $O(N^2)$ space.
There is an implementation of the dynamic programming version both in C and in Pascal.

3. Running times

The following table sums up the running times of our programs, for the input polygons to be used in the evaluation. For the table to fit on the page, we abbreviated the sentence "it runs in less than one second" to "< 1 sec", whereas "> 600 sec" means that the program has been aborted after ten minutes. There are no significant speed differences between the corresponding Pascal and C programs.

Algorithm	INPUT 1	INPUT 2	INPUT3	INPUT 4	INPUT5
Naive — EXP	< 1 sec	< 1 sec	< 1 sec	> 600 sec	> 600 sec
Divide and Conquer — EXP	< 1 sec	< 1 sec	< 1 sec	> 600 sec	> 600 sec
Memory Function — $O(N^4)$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 3 sec
Dynamic Programming — $O(N^4)$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 2 sec
Dynamic Programming — $O(N^3)$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec