Programming Problem: Matrix Chain Multiplication

[14 points for the programs, 6 points for the analysis.]

For this problem, you get to implement three (simple) algorithms for the matrix chain multiplication problem. Like the dynamic programming algorithm for this, we won't actually be computing the product of a matrix chain; we'll be computing the cost of multiplying the chain of matrices with different strategies for computing intermediate results (i.e., different placements of the parentheses).

Implementation Language

As with previous assignments, you get to implement your solution in Java, C or C++, whichever language you prefer. The name of your source files will depend on your language:

- For Java : naive.java, greedy.java and dp.java
- For C : naive.c, greedy,c and dp.c
- For C++ : naive.cpp, greedy.cpp and dp.cpp

When we test your solutions, we'll compile and run them on an EOS Linux machine using the following commands. Be sure to try these out yourself to make sure you program will compile and work when we try it out. Your program will read its input from standard input (i.e., from the terminal), but, like in the following examples, we'll use I/O redirection to get it to read from a file instead.

- For Java

```
javac naive.java
java naive < input-3.txt

javac greedy.java
java greedy < input-3.txt

javac dp.java
java dp < input-3.txt
```

- For C

```
gcc -Wall -std=c99 -g -O2 -o naive naive.c
./naive < input-3.txt

gcc -Wall -std=c99 -g -O2 -o greedy greedy.c
./greedy < input-3.txt

gcc -Wall -std=c99 -g -O2 -o dp dp.c
./dp < input-3.txt
```

- For C++

```
g++ -Wall -g -O2 -o naive naive.cpp
./naive < input-3.txt

g++ -Wall -g -O2 -o greedy greedy.cpp
./greedy < input-3.txt

g++ -Wall -g -O2 -o dp dp.cpp
./dp < input-3.txt
```

Input Format

All the programs will read the description of a matrix chain multiplication problem from standard input. The input will start with a value, n, the number of matrices in the chain. This will be followed by $n + 1$ integers, $p_0 \dots p_n$ giving the dimensions of each matrix in the chain. Matrix M_i (for $1 \leq i \leq n$) will be p_{i-1} rows tall and p_i columns wide. Values are separated by whitespace, and the $p_0 \dots p_n$ sequence may be split across several lines to make it easier to look at in a text editor.

The following shows sample input, `input-1.txt`. This describes the product of 8 matrices. The first matrix is 8×10, the next one is 10×4, the next is 4×9 and so on.

```
8
8 10 4 9 1 3 5 9 4
```

All the matrix dimensions will be between 1 and 10^4, inclusive. This will let us consider large matrix chain multiplication problems, where it might be very important to spend a little extra time to figure out the best way to compute the product before staring to actually multiply matrices.

Note that the matrix dimensions will fit in a signed, 32-bit integer, but the total cost of multiplying the matrices will need a wider type. Use a signed 64-bit integer for this (the `long` type in Java and C and C++ on an EOS Linux machine).

Program Output

As output, just print one line giving the total cost of multiplying the matrix chain. For the cost, we'll just count the total number of scalar multiplications that need to be performed while multiplying matrices (so, we're ignoring the cost of additions or other costs like iterating over the rows and columns of the matrices). Like we did in class, we'll assume matrices will be multiplied using the basic $\Theta(H_1 * W_1 * W_2)$ matrix multiplication algorithm (where the left-hand matrix is $H_1 \times W_1$ and the right-hand matrix is $W_1 \times W_2$).

The total cost you get will depend on the program's strategy for placing the parentheses, so it will typically vary across the three programs your writing. Remember that the total cost might be a big number, so you should compute and print it using a signed 64-bit integer.

Naive Strategy, 4 pts

Your `naive` program will compute and report the cost of multiplying the matrix chain left-to-right. So, it will assume matrix M_1 will be multiplied by M_2. Then, that result will be multiplied by M_3, that result will be multiplied by M_4 and so on.

Greedy Strategy, 5 pts

your `greedy` program will use a recursive, greedy approach for deciding how to compute intermediate matrix products. At the top-level call, it will decide the last multiplication operation to perform. It will do this by finding the value k that minimizes the cost of multiplying the result of $M_1 \cdot M_2 \cdot \dots M_k$ times the result of $M_{k+1} \cdot M_{k+2} \cdot \dots M_n$. So, it's just trying to minimize the cost of this last multiply operation, without (yet) considering the cost of computing the product on the left $(M_1 \cdot M_2 \cdot \dots M_k)$ or the product on the right $(M_{k+1} \cdot M_{k+2} \cdot \dots M_n)$.

Once the greedy algorithm has chosen the value of k that minimizes this cost, it will make recursive calls to determine how to compute the product on the left, $M_1 \cdot M_2 \cdot \dots M_k$ and how to compute the product on the right, $M_{k+1} \cdot M_{k+2} \cdot \dots M_n$. These recursive calls will continue down to a base case of just one matrix.

Note that this greedy algorithm might give us a fairly good strategy for multiplying a particular chain of matrices, but, in general, it's not guaranteed to be optimal. You can see this from the first sample input. Recursively choosing how to multiply the chain like this isn't good example of a greedy algorithm. It's not guaranteed to give us an optimal solution, and even after choosing a value of k. Also, after greedily choosing a value for k, we still have two subproblems to solve.

DP Strategy, 5 pts

Your `dp` program will will implement the dynamic programming technique for matrix chain multiplication discussed in class and in the textbook. This should give an optimal cost for multiplying the chain of matrices.

If it's helpful in debugging your solution, there's a YouTube video showing the dynamic programming solution on `input-1.txt`. It shows how each subproblem is being solved as the algorithm works its way from smaller problem instances to larger ones.

`https://youtu.be/ifcjL7rd6Rw`

Sample Execution

We're providing five sample inputs with this assignment. There are some smaller ones to help you check the behavior of your program and some larger ones to help you see a little bit of the performance costs of different techniques. If you run your program on these inputs, here's what you should expect (shown here for a Java):

```
$ java naive < input-1.txt
1472
$ java greedy < input-1.txt
672
$ java dp < input-1.txt
284

$ java naive < input-2.txt
438000
$ java greedy < input-2.txt
3630000
$ java dp < input-2.txt
438000

$ java naive < input-3.txt
3630000
$ java greedy < input-3.txt
3630000
$ java dp < input-3.txt
438000

$ java naive < input-4.txt
20375128344
$ java greedy < input-4.txt
7053967329
$ java dp < input-4.txt
51646670
```

Algorithm Analysis

Write up a short, worst-case analysis of each of your algorithms for matrix chain multiplication. We're not looking for the cost of multiplying the chain of matrices (our programs don't really multiply the matrices); we're looking for the cost running each of the three algorithms. Given the dimensions of a chain of n matrices, what's the cost of determining how the chain should be multiplied and computing the associated, total number of scalar multiplies entailed?

In your analysis, include a section for each of the three algorithms. In each section, give a brief explanation of your analysis, leading to the worst-case asymptotic runtime for the algorithm. It should take between 1 and 1.5 pages to report your analysis of all of the algorithms.

Submitting Your Work

In Moodle, you'll see an assignment named `Assignment 3`. Submit the source code to your programs using this link. For your analysis, submit that to a GradeScope assignment with the name `Program 3 Analysis`.