EX10

1. Introduction. This is a literate program which solves the problem set in lab ten—Matrix Chain Multiplication.

This problem is stated as follows:

Given a sequence of N commensurable matrices, $A_1, A_2 \ldots A_N$, determine the optimal order in which to compute the product $A_1 \times A_2 \times \ldots \times A_N$.

We will start with the, by now familiar, outline of the program. We don't need any functions other than main and there will be no global data either.

Actually, that is not quite true. We will need one function for a small bonus at the end of our program.

⟨ Headers 4 ⟩

using namespace std;

⟨ Prototypes for functions 16 ⟩
⟨ The main program 2 ⟩
⟨ Implementation for functions 17 ⟩

2. Main. OK, let's start writing *main*. The skeleton of the *main* program is as follows.

```
⟨ The main program 2 ⟩ ≡
  int main( )
  {
    ⟨ Variables of main 3 ⟩
    ⟨ Open the input file 5 ⟩
    ⟨ Read in the matrix dimensions and initialize the data 7 ⟩
    ⟨ Do the main loop 10 ⟩
    ⟨ Finish and clean up 14 ⟩
  }
```

This code is used in section 1.

3. The first thing we need to do is declare the variables we need to input data. Let's start with the character array *filename* and the input stream *fin*.

```
⟨ Variables of main 3 ⟩ ≡
  char filename[20];
  ifstream fin;
```

See also sections 6, 8, and 9.

This code is used in section 2.

4. We need a couple of header files for handling the input process:

```
⟨ Headers 4 ⟩ ≡
#include <iostream>
#include <fstream>
```

See also section 11.

This code is used in section 1.

5. Right—now we can get the file opened, ready for input. We will prompt for the input file name using *cerr* so that we can redirect the output without getting the prompt in the output file and so that we can see the prompt even when we redirect standard output. We will then read in the file name and open an input stream. We should test for errors too, I guess.

```
⟨ Open the input file 5 ⟩ ≡
  cerr ≪ "Please␣enter␣the␣name␣of␣the␣input␣file:␣";
  cin ≫ filename;
  fin.open(filename);
  if (¬fin) {
    cerr ≪ "Error␣opening␣file␣" ≪ filename ≪ ".␣Program␣will␣exit." ≪ endl;
    return 0;
  }
```

This code is used in section 2.

6. Sanity checking. Before we start looping over the input we need to read in the matrix dimensions from the input file.

The input file contains an integer variable N, the number of matrices to be mutiplied, followed by N pairs of row and column dimensions.

The sequence of matrices $A_1, A_2, \ldots, A_N$ can only be multiplied if the number of rows in each matrix is identical to the number of columns in its predecessor. Formally, $\forall\ i;\ 1 < i \leq N : row_i = col_{i-1}$. This means that, rather than storing N pairs of dimensions, we need only store $N + 1$ dimension values. We will store these in the array *size*.

```
⟨ Variables of main 3 ⟩ +≡
  int N;
  int *size;
```

7. We can now read the data. As we read in the pairs we will use the requirement that dimensions match from matrix to matrix as a sanity check on the input.

```
⟨ Read in the matrix dimensions and initialize the data 7 ⟩ ≡
  fin ≫ N;
  size = new int[N + 1];
  for (int i = 1; i ≤ N; i++) {
    fin ≫ check ≫ size[i];
    if (i ≡ 1) size[0] = check;
    else {
      if (size[i − 1] ≠ check) {
        cerr ≪ "Error:␣matrices␣are␣incommensurable!" ≪ endl;
        cerr ≪ "␣␣␣␣␣Matrix␣" ≪ i − 1 ≪ "␣has␣" ≪ size[i − 1] ≪ "␣columns,␣and␣" ≪ endl;
        cerr ≪ "␣␣␣␣␣matrix␣" ≪ i ≪ "␣has␣" ≪ check ≪ "␣rows." ≪ endl;
        return 1;
      }
    }
  }
```

This code is used in section 2.

8. We need a variable to hold the *check* value, the number of rows of each matrix.

```
⟨ Variables of main 3 ⟩ +≡
  int check;
```

9. DP Matrix multiplication. Now we can proceed with the main part of the algorithm.

The approach we are going to take is a bottom up computation of the best way to multiply out all possible partitions of the matrix sequence.

We will iterate over the number of matrices in the sub-sequence, increasing the length of of the chain from 1 up to N. At each length we will iterate across all possible ways of splitting the chain into two pieces.

The array $best[\,][\,]$ will hold the lowest cost associated with multiplying together sub-sequences of matrices. Thus, $best[i][j]$ will contain the lowest cost for evaluating the product $A_i \times A_{i+1} \times \ldots \times A_{j-1} \times A_j$.

```
⟨ Variables of main 3 ⟩ +≡
  int **best;
  int **split;
```

10. There are a couple of preliminary steps before the main loop; creating the $best[\,][\,]$ array and initializing it. Once this has been done we can proceed to the main iterative calculation.

We note that array A_i has $size[i-1]$ rows and $size[i]$ columns.

If we wish to multiply all matrices from A_i up to A_k we can do this in a number of ways. Essentially we can vary j over all values from i up to $k-1$ and, provided we already have the best way to calculate the sub-products $A_i \times A_{i+1} \times \ldots \times A_j$ and $A_{j+1} \times \ldots \times A_{k-1} \times A_k$, then we can readily evaluate the total cost for each choice of k.

This cost has three components:

- the minimum cost of evaluating $A_i \times A_{i+1} \times \ldots \times A_j$, stored in $best[i][j]$;
- the minimum cost of evaluating $A_{j+1} \times \ldots \times A_{k-1} \times A_k$, stored in $best[i][j]$;
- the cost of performing the final multiplication.

The cost for this partition is simply the sum of these three components.

```
⟨ Do the main loop 10 ⟩ ≡
  ⟨ Declare the best array 12 ⟩
  ⟨ Intitialize the best array 13 ⟩
  for (int chainLength = 2; chainLength ≤ N; chainLength++) {
    for (int i = 1; i ≤ N − chainLength + 1; i++) {
      int k = i + chainLength − 1;
      best[i][k] = INT_MAX;
      for (int j = i; j ≤ k − 1; j++) {
        int cost = best[i][j] + best[j + 1][k] + size[i − 1] ∗ size[j] ∗ size[k];
        if (cost < best[i][k]) best[i][k] = cost;
        split[i][k] = j;
      }
    }
  }
```

This code is used in section 2.

11. We need a header file for `INT_MAX` to be defined.

```
⟨ Headers 4 ⟩ +≡
#include <climits>
```

12. We create the double dimensioned array *best* in two stages. First, we create an array of **int** $*$ pointers which will address the rows of *best*. Then, for each row, we will create an integer array which will hold the column entries for the corresponding row.
Note: The *best* array will be declared with $N+1$ rows and columns so that we can store the solution for the sub-sequence from A_i to A_k in $best[i][k]$. This means that entries of the form $best[0][i]$ and $best[i][0]$ are not used. (Actually, only half of the entries in *best* are ever used, entries of the form $best[i][j]$ where $i > j$ never get set or examined, but finding an optimal way to store a triangular matrix is outside the scope of this subject.)

```
⟨ Declare the best array 12 ⟩ ≡
  best = new int*[N + 1];
  split = new int*[N + 1];
  for (int i = 0; i ≤ N; i++) {
    best[i] = new int[N + 1];
    split[i] = new int[N + 1];
  }
```

This code is used in section 10.

13. The starting point for our bottom up solution is the set of N individual matrices. As there is no multiplication involved with a single matrix, the associated cost will be 0.

```
⟨ Intitialize the best array 13 ⟩ ≡
  for (int i = 1; i ≤ N; i++) {
    best[i][i] = 0;
    split[i][i] = i;
  }
```

This code is used in section 10.

14. Now that we have finished our search we can report the results and clean up. The least computationally costly solution to multiplying all the matrices is simply the value of best[1][N].

```
⟨ Finish and clean up 14 ⟩ ≡
  cout ≪ "Minimum␣cost␣=␣" ≪ best[1][N] ≪ endl;
  fin.close( );
```

See also section 15.

This code is used in section 2.

15. Bonus. You may have noticed several references to the array *split*[ ][ ] in the preceding code. This has been used to track the optimal partition corresponding to the minimal cost stored in *best*[*i*][*j*]. We can use this array to reconstruct and display the optimal sequence.

To do this we will use a recursive function *show*( ) to display the sequence. The output from *show* is a fully parenthesized sequence where matching parentheses indicate the optimal sub-sequences. Thus, output of the form `"((1)(2))(3)"` would indicate that the best result comes from multiplying A_1 by A_2 and then multiplying this result by A_3.

We kick this off by calling as follows:

```
⟨ Finish and clean up 14 ⟩ +≡
  cout ≪ "The␣optimal␣multiplication␣sequence␣is␣as␣follows:" ≪ endl;
  show(split, 1, N);
  cout ≪ endl;
```

16. First is the prototype:

```
⟨ Prototypes for functions 16 ⟩ ≡
  void show(int **, int, int);
```

This code is used in section 1.

17. And then the implementation: This function determines the point at which the input sequence $A_{start} \ldots A_{end}$ is split for optimal evaluation. Thus, to evaluate $A_{start} \times \ldots \times A_{end}$ we multiply $A_{start} \times \ldots \times A_{mid}$ by $A_{mid+1} \times \ldots \times A_{end}$. The value *mid* is exactly what was stored in *split*[*start*][*end*]. When we are down to a single matrix, we output its index. Otherwise, we parenthesize the recursive calls for the left and right sub-sequences.

```
⟨ Implementation for functions 17 ⟩ ≡
  void show(int **split, int start, int end)
  {
    if (start ≡ end) cout ≪ start;
    else {
      int mid = split[start][end];
      cout ≪ "(";
      show(split, start, mid);
      cout ≪ ")";
      cout ≪ "(";
      show(split, mid + 1, end);
      cout ≪ ")";
    }
    return;
  }
```

This code is used in section 1.

18. Index. This index is automatically created. It lists all the variables used in the program and the section(s) in which they are used. Underlined entries indicate where a variable is defined. The remaining sections of this document are also created automatically.

⟨ Declare the *best* array 12 ⟩ Used in section 10.
⟨ Do the main loop 10 ⟩ Used in section 2.
⟨ Finish and clean up 14, 15 ⟩ Used in section 2.
⟨ Headers 4, 11 ⟩ Used in section 1.
⟨ Implementation for functions 17 ⟩ Used in section 1.
⟨ Intitialize the *best* array 13 ⟩ Used in section 10.
⟨ Open the input file 5 ⟩ Used in section 2.
⟨ Prototypes for functions 16 ⟩ Used in section 1.
⟨ Read in the matrix dimensions and initialize the data 7 ⟩ Used in section 2.
⟨ The main program 2 ⟩ Used in section 1.
⟨ Variables of main 3, 6, 8, 9 ⟩ Used in section 2.