Computer science problems.

About the problems. The theme of this year's problems is dynamic programming, which is a technique to eliminate overhead of repeated recomputing of the same subproblems that results from recursive definitions.

What you need to do. For these problems we ask you to write a program (or programs), as well as write some "paper-and-pencil" solutions (use any text editor that you see fit, or scan an actual handwritten solution; convert the result to pdf format if possible). You may use any programming language you want for your programs, as long as its full implementation is available at no cost and with an easy installation for both a Mac and Windows. It is best to implement each problem as a separate function so that we can run them separately. We will be looking for the following in your submissions:

- Correct code that we can run. You need to send us all your code files, including the header files for languages like C++. If you are using standard libraries, make sure to include all "import" statements, as required by the language you are using. Make sure to send the files under the correct names, including the file extension (.java, .c, etc). Make sure that the file names do not contain any identifying information about you, such as your first or last name.

- Test data for your code that you have used (you can write it in comment or in a separate file). Make sure to test your code well – you don't want it to fail our tests!

- Code documentation and instructions. **Important: do not include your name in comments or in any file names.** If you are submitting your answers to non-code problems in a separate file, also make sure that it does not have your name in the contents or in the file name. The only place where you specify your name is the zip file with your solutions which must be of the form `yourlastname-CS-solution.zip` (replace `yourlastname` by your actual last name). **Make sure that you use zip compression, and not any other one, such as tar**. In the beginning of each file specify, in comments:

  1. Problem number(s) in the file. If you have a file with "helper" functions, mark it as such.
  2. The *programming language*, including the *version* (Java 1.7 or 1.8, for instance), the *development framework* (such as Visual Studio) that you used, unless you were using just a plaintext editor (notepad, emacs, etc), and the *platform* (such as Windows, Mac, Linux)
  3. Instructions for running your program (how to call individual functions, pass the input (if any), etc), either in comments in your program file or as a separate file, clearly named. Your program may get input from the user (i.e. it asks to enter some data and then reads it) or you may store the data in specific variables within your program. You need to clearly explain how to input or set the data.

4. Some of your code may be commented out if it is not used in the final run of your program. Make sure it is clear what needs to be uncommented to run code for each of the problems.
5. All of your test data.
6. If you were using sources other than the ones listed here (i.e. textbooks, online resources, etc) for ideas for your solutions, please clearly credit these contributions. This is a courtesy to work of others and a part of ethics code for scholars.

- Clear, understandable, and well-organized code. This includes:

1. Clear separation between problems; comments that help find individual problems and explain how to run the corresponding functions.
2. Breaking down code into functions that are clearly named and described (in comments), using meaningful names for variables and function parameters. Your code should be as self-explanatory as possible. While using comments helps, naming a variable `average` is better that naming it `x` and writing a comment "x represents the average".
3. Minimization of code repetition. Rather than using a copy-paste approach, use functions for repeated code and reuse these functions.
4. Using well-chosen storage structures (use an array or a list instead of ten variables, for instance) and well-chosen programming constructs (use loops or recursion when you can, rather than repeated code).
5. While we are not asking for the fastest program (it's better to make it more readable), you should avoid unnecessary overhead.

Problem 1. Fibonacci numbers are a well-known sequence of numbers defined as follows:

$$\begin{aligned} F_1 &= 1, \\ F_2 &= 1, \\ F_n &= F_{n-1} + F_{n-2} \text{ for all } n > 2. \end{aligned}$$

1. What is F_8?
2. Write a function that takes a single positive integer n and returns F_n. Your function **must** implement the definition directly, i.e. it must be making two recursive calls for every $n > 2$. As for all problems, please include tests for your function.
3. How many function calls does your function make in order to compute F_5? What are those function calls?
4. Explain how the number of these calls would grow when n increases. Why is this not a reasonable way of computing Fibonacci numbers?

Problem 2. We use the example of Fibonacci numbers to introduce approaches that will be used later in the problem set. While there are other, more direct, ways of computing Fibonacci numbers, we will explore optimizations that we can make based on just the definition by eliminating the overhead of unnecessarily repeated function calls.

One of the approaches is known as *memoization* and is based on storing results of function calls the first time the function is called with a given set of parameters, and then using the stored results instead of recomputing them.

More specifically, the program maintains a table of calls to a given function. The table contains the parameter(s) and the result. Every time the function is called, a lookup is performed to see if the table already contains the results for given parameter(s). If it does, the result from the table is returned. Otherwise the computation proceeds (although recursive calls then may perform their own lookups). Once the result is computed, it is added to the table and returned from the function.

1. For this problem, please write a new version of the Fibonacci function that performs memoization. As before, make sure to test it thoroughly. You may use any data structure of your choice to maintain the lookup table. However, even if your programming language supports automated memoization, you have to implement it on your own.

2. How many function calls does your program make for $n = 5$? For $n = 100$? Please explain your answer.

Problem 3. You may have noticed that the table of results in memoization is being filled in the increasing order of function parameters: first $n = 1$, then for $n = 2$, and so on. That is because the results for higher numbers depend on those of the lower ones. This indicates that instead of performing a sequence of function calls top-down (from higher numbers to lower ones), one can compute Fibonacci numbers bottom-up (from F_1 and F_2 up to F_n) in a loop. The loop will require two variables to keep two most recently computed Fibonacci numbers. Each next iteration will replace the current F_k and F_{k+1} numbers with F_{k+1} and $F_{k+2} = F_k + F_{k+1}$, respectively. The loop will start with F_1 and F_2 and continue until the n-th Fibonacci number is computed.

Write a function that takes n and returns F_n. The function must have only a loop, no recursion or a table.

Problem 4: optimal matrix multiplication problem. *Notation: in this problem we use $\times$ for matrix multiplication and $\cdot$ for multiplication of numbers (also referred to as scalar multiplication).*

The following problem is described in Chapter 15 of [1] in more detail. In physics and graphics processing one often needs to multiply a large number of matrices of various dimensions. Recall that the product of an $n \times m$ (where n is the number of rows, m is the number of columns) matrix A and an $m \times k$ matrix B is an $n \times k$ matrix C with entries $c_{i,j} = \sum_{l=1}^{k} a_{i,l} b_{,lj}$, where $a_{i,l}$ and $b_{l,j}$ refer to the elements of the matrices A and B respectively. Thus, in order to produce one

element of C one needs to perform m *scalar* multiplications, i.e. multiplications of individual elements. The total number of scalar multiplications to compute C is $n \cdot m \cdot k$.

The product of two matrices A and B is defined only if the number of columns of A is the same as the number of rows of B. We call such matrices *compatible*.

Consider multiplying matrices $A_1 \times A_2 \times A_3 \times A_4$ that have the following dimensions:

$$A_1 : 5 \times 4,$$
$$A_2 : 4 \times 2,$$
$$A_3 : 2 \times 10,$$
$$A_4 : 10 \times 3.$$

1. Suppose one performs the multiplications left-to-right: $((A_1 \times A_2) \times A_3) \times A_4$. How many scalar multiplications would be performed?

2. How many scalar multiplications are performed when the multiplication proceeds right-to-left: $A_1 \times (A_2 \times (A_3 \times A_4))$?

Briefly explain your answers.

Problem 5. Since not all ways of multiplying matrices require the same number of scalar multiplications, we will consider the following problem. The *matrix chain multiplication problem* is defined as follows: given a sequence of matrices $A_1, A_2, \ldots A_n$, where A_i has dimensions $p_{i-1} \times p_i$, and A_{i-1} and A_i are compatible for all $1 \leq i \leq n$, determine the order of computing $A_1 \times A_2 \times \cdots \times A_n$ with the minimum number of scalar multiplications. Note that the problem does not ask you to multiply the matrices, just to figure out the optimal way of multiplying them.

Let $m[i, j]$ denote the optimal (minimal) number of scalar multiplications to multiply matrices from an index i to an index j ($1 \leq i \leq j \leq n$) in a given matrix sequence. $m[i, j]$ is defined recursively as follows:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j\} & \text{if } i < j \end{cases}$$

Apply this formula to determine the optimal way of multiplying $A_1 \times A_2 \times A_3$ (i.e. the case when $i = 1, j = 3$) in the example in the previous problem. Also determine the optimal way of multiplying $A_2 \times A_3 \times A_4$. Show your work.

Problem 6. One may use the formula in Problem 5 to construct a recursive solution for the matrix chain multiplication problem, where each computation of $m[i, j]$ is implemented as its own function call, with the case $i = j$ acting as the base case. How many function calls does one need to make to compute $m[1, 4]$? How many of these calls are with parameters for which the answer has been previously computed?

Now consider a problem with 5 matrices. How many calls (total) one would make, and how many of these calls repeat parameters of previously computed calls?

If one were to use a memoization approach, how many function calls would they be making for $n = 4$ and $n = 5$?

Please explain your answers. You don't need to implement a recursive solution or a solution with memoization.

Problem 7. The previous problem indicates that a naive recursive solution results in repeated calls to the function that computes $m[i, j]$ for the same values of i and j. These repeated calls constitute *overlapping subproblems*. While it is possible to use memoization to minimize the number of function calls, there is also a bottom-up approach that builds the solutions from smaller ones to larger ones in a table, similar to how Fibonacci are computed in a loop in Problem 3. A bottom-up approach to an optimization problem with overlapping subproblems is known as *dynamic programming*.

In the case of matrix chain multiplication the solution is constructed by filling in the following table that shows values of $m[i, j]$ for all $1 \leq i \leq j \leq n$. The table is filled up bottom-up, applying the equation given in Problem 5. The bottom row represents the number of scalar multiplications $m[i, i]$ performed in "multiplying" a sequence of just one matrix (which is the base case of the definition, with the value 0). The second row consists of the optimal costs (the number of scalar multiplications) in multiplying pairs of neighboring matrices, so it represents values $m[1, 2], m[2, 3], \ldots$. The third row from the bottom represents the optimal costs of multiplying groups of three matrices: $m[1, 3]$, which is the optimal cost of multiplying $A_1 \times A_2 \times A_3$, then $m[2, 4]$, which is the optimal cost of multiplying $A_2 \times A_3 \times A_4$. The top row has only one element that contains the solution for the entire sequence, from the first matrix to the last one.

For instance, for a sequence of four matrices the table is as follows:

$$
\begin{array}{ccccccc}
 & & & m[1,4] & & & \\
 & & m[1,3] & & m[2,4] & & \\
 & m[1,2] & & m[2,3] & & m[3,4] & \\
m[1,1] & & m[2,2] & & m[3,3] & & m[4,4]
\end{array}
$$

Consider the dimensions of the matrices given in Problem 4. We start filling in the table bottom-up for this case:

$$
\begin{array}{cccccc}
 & & & ? & & \\
 & & ? & & ? & \\
 & 40 & & 80 & & 60 \\
0 & & 0 & & 0 & & 0
\end{array}
$$

The bottom row of the table represents the base case which is all 0s. The second row represents the number of multiplications in multiplying two matrices: the first cell is the cost of multiplying $A_1 \times A_2$ which is $5 \cdot 4 \cdot 2 = 40$ scalar

multiplications, the second cell corresponds to $A_2 \times A_3$ which is $4 \cdot 2 \cdot 10 = 80$, and the last cell is the cost of $A_3 \times A_4$ which is $2 \cdot 10 \cdot 3 = 60$. Note that there is no choice involved in filling in the bottom two rows of the table.

The third row contains $m[1,3]$ and $m[2,4]$ that you have computed in Problem 5. Fill in these values. Then compute $m[1,4]$ (the cost of the optimal way of multiplying all four matrices) using the equation in Problem 5. Show your work. Also show the parentheses in the sequence of four matrices that group the matrices in the way that produces the optimal solution.

Problem 8. Implement the dynamic programming solution (the one described in the previous problem) for any sequence of matrices. Specifically, your program must take input given as n (the number of matrices in the sequence) and the sequence of $n+1$ numbers as dimensions $p_0, p_1, \ldots p_n$ (the matrices assume to be compatible). The program must produce the total number of scalar multiplications in the optimal solution, as well as the full set of parentheses in the solution, for instance:

$$(A1 * ((A2 * A3) * A4))$$

Here $*$ stands for multiplication (since most programming languages cannot represent $\times$). You may skip, or include, the outer set of parentheses.

Your program must implement dynamic programming, not memoization.

If two or more ways of putting parentheses produce the same result, either one can be produced by your program.

Please test your program on the sequence of 5 matrices with the dimensions as follows: $30, 20, 10, 40, 5, 25$. Submit your results (the minimal number of scalar multiplications and the parentheses for the case when it happens).

Problem 9: breaking up a string. Now consider the following problem: A certain string processing language allows the programmer to break a string into two pieces. Since this involves copying the old string into two new locations, it costs n units of time to break a string of n characters into two pieces. There is no operation that allows you to break a string into more than two pieces at once. For instance, breaking a string into three pieces requires breaking it into two pieces twice (with the corresponding copying). Suppose a programmer wants to break a string into many pieces. The order in which the breaks are made can affect the total amount of time used. For example suppose we wish to break a 20 character string after characters 2,8, and 10:

- If the breaks are made in left-right order, then the first break costs 20 units of time, the second break costs 18 units of time and the third break costs 12 units of time, a total of 50 steps.
- If the breaks are made in right-left order, the first break costs 20 units of time, the second break costs 10 units of time, and the third break costs 8 units of time, a total of only 38 steps.

Come up with an optimal solution (by hand) for breaking a string of 20 units after the characters 2, 8, and 10. Show all your work.

Problem 10. Describe a recursive solution for this problem. It must contain a recursive definition, similar to the one in Problem 5. Make sure to define your notations; don't forget about the base case.

Explain what overlapping subproblems arise in the recursive solution.

Problem 11. Describe a dynamic programming solution as a table that gets filled up in a bottom-up fashion. Fill in the table for the example in Problem 9.

Implement your solution using dynamic programming or memoization. Make sure that your solution shows both the total number of copied characters and the order of the positions at which the string is broken.

Your program must take the length of the string and the sequence of break positions as inputs. It should output the total number of copied characters in the optimal solution, as well as the order of positions at which the string is broken. The example format of the sequence is $2, 10, 5$ indication that first the string is broken after the second character, then after the 10th character of the original string, and then after the 5th character of the original string.

Submit your results for the example in Problem 9.

Problem 12. Suppose a friend tells you that there is no need in constructing a bottom-up solution since the optimal number of copies can always be achieved by choosing the break closest to the middle in a top-down fashion. For instance, if a string of 20 units needs to be broken after characters 2, 5, 11, and 17, then choosing the break at 11 (closest to 10, which is the middle) would lead to the best solution.

Do you agree that this strategy guarantees the best solution? If yes, please clearly explain why. If not, please show an example for which this is not the case.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.