Matrix chain multiplication

Marcos Crespo Díaz

2024-03-17

Introduction

The aim of this document is to solve the four tasks proposed as deliverables for the course *High Performance Computing for Data Science* of the *MSc Statistics for Data Science* at *UC3M*. The assignment focuses on the problem of chain matrix multiplication and associativity. The objective of this course is to find intelligent and efficient solutions using different tools that the R environment provides us to solve computationally complicated or demanding problems.

About the assignment

The matrix chain multiplication problem addresses the optimization problem of finding the most efficient sequence for multiplying a given series of matrices. The objective isn't to execute the multiplications themselves but rather to determine the order in which these matrix multiplications should occur.

This efficiency can be measured both in number of required operations[1] or in computation time[2].

Matrix chain multiplication is a subject of study for High Performance computing. The computational challenge arises from the associative property of matrix multiplication. This property leads to multiple ways of computing the product of matrices (with the same result), resulting in variations in number of operations needed to perform the multiplication. If the difference in operations is very big from one parenthesization to another, then the computational cost may be very different.

Determining the best parenthesization is not an easy task, since the number of possible parenthesizations of a set of matrices is exponential[3].

The tasks that have to be solved are four:

- Task 1: Find a compelling example of the multiplication of 5 matrices in which the default ordering[4] takes considerably more time than some custom parenthesization. Show this with a benchmark. Analyse the number of operations for each case and discuss the results.
- Task 2: Implement a C++ function that takes 5 matrices as input and multiplies them using library RcppArmadillo without any parenthesization. Discuss the results.
- Task 3: Implement a C++ function that takes a list of matrices

[1] Note that given two matrices of dimensions $\mathcal{X} \times \mathcal{Y}$ and $\mathcal{Y} \times \mathcal{Z}$ it takes $\mathcal{X} * \mathcal{Y} * \mathcal{Z}$ operations to compute the result

[2] Time elapsed between the beginning of the process and when it finishes

[3] The number of ways to parenthesize an expression of $n+1$ terms is given by the n^{th} Catalan number $C_n = \frac{1}{n+1}\binom{2n}{n}$

[4] `A%*%B%*%C%*%D%*%E`

and a vector defining the multiplication order, and performs the operations in the specified order using RcppArmadillo. Compare it with the previous case for the best parenthesization.

- Task 4: Investigate further in the scientific literature about the matrix chain ordering problem, specifically about the available algorithms. Implement at least one algorithm that solves the problem and produces a vector defining the multiplication order. Use it in conjunction with the function from task 3 to compare the approach of solving the matrix chain ordering problem and perform the product in the resulting order vs. R's best and worst cases (task 1) vs. Armadillo's naive approach (task 2).

Methodology

In the resolution of this assignment we will be using several R libraries and functionalities. First, in order to measure the performance of the different functions and operations we compare, we will be using `microbenchmark` [Mersmann, 2023] package with 1000 repetitions of each operation and compare the means[5].

[5] This is the $\frac{1}{n}\sum t_i$ of all the execution times t_i for each process

Which function is the fastest will be easily visualize using custom bar plots.

Finally, in order to speed up the computations we will be using `Rcpp`[Eddelbuettel and Balamuta, 2018]. Rcpp is a package for R that facilitates integration of C++ code. It allows for seamless calling of C++ functions from R, enhancing performance thanks to the compiled nature of the language. This will be very useful.

Furthermore, matrix are typically critical objects when programming, because matrix operations are normally computationally expensive[6]. Several libraries exist in order to enhance the computation of matrix multiplications and we will be using `RcppArmadillo` [Eddelbuettel and Sanderson, 2014] library.

[6] Matrix chain multiplication is an example

Advantages of Using Armadillo

- **Concise Syntax**: Armadillo simplifies the syntax for matrix operations, making the code more readable and maintainable.
- **Efficiency**: Armadillo leverages optimized algorithms for high-performance numerical computing.
- **Integration with Rcpp**: Rcpp facilitates seamless integration between R objects and C++ data structures, allowing efficient computation with Armadillo within an R environment.

This approach offers a powerful and efficient solution for matrix multiplication tasks within R using C++.

Task 1

Given a set of matrices $\mathcal{A}_{50\times15}$, $\mathcal{B}_{15\times70}$, $\mathcal{C}_{70\times10}$, $\mathcal{D}_{10\times50}$, $\mathcal{E}_{50\times2}$; the naive multiplication $\mathcal{ABCDE}$ takes 1.175×10^5 operations[7].

Now, lets consider some different parenthesization, for example $\mathcal{A}(\mathcal{BC})\mathcal{DE}$. The number of operations involved are now 4.8×10^4.[8] This is a reduction of nearly 59.1489362% of the complexity compared to the naive approach.

Furthermore, lets consider now the order $\mathcal{A}((\mathcal{BC})(\mathcal{DE}))$. Now the number of operations involved are 1.4×10^4.[9] This time the reduction is 88.0851064% compared to the standard approach but also 70.8333333% compared to the second approach.

Lets create some virtual matrices for visualizing this results:

[7] $50 * 15 * 70 + 50 * 70 * 10 + 50 * 10 * 50 + 50 * 2 * 50$

[8] $15 * 70 * 10 + 50 * 15 * 10 + 50 * 10 * 50 + 50 * 50 * 2$

[9] $15 * 70 * 10 + 10 * 50 * 2 + 10 * 50 * 2 + 50 * 15 * 2 =$

```
A <- matrix(rnorm(50 * 15), nrow = 50, ncol = 15)
B <- matrix(rnorm(15 * 70), nrow = 15, ncol = 70)
C <- matrix(rnorm(70 * 10), nrow = 70, ncol = 10)
D <- matrix(rnorm(10 * 50), nrow = 10, ncol = 50)
E <- matrix(rnorm(50 * 2), nrow = 50, ncol = 2)
```

```
res0 <-  microbenchmark::microbenchmark(A %*% B %*% C %*% D %*% E,
                                        A %*% (B %*% C) %*% D %*% E,
                                        A %*% ((B %*% C) %*% (D %*% E)),
                                        times = 1000)
```

Table 1: First Benchmark

expr	min	lq	mean	median	uq	max	neval
A %% *B* %% C %% *D* %% E	72.5	82.5	93.5356	84.2	92.70	4147.7	1000
A %% *(B* %% C) %% *D* %% E	30.4	34.9	38.5222	35.8	38.05	126.3	1000
A %% *((B* %% C) %% *(D* %% E))	8.6	9.6	10.9509	10.0	11.00	35.6	1000

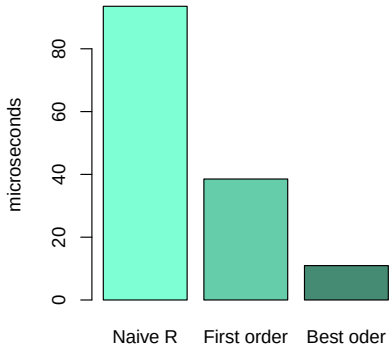

Figure 1: Benchmark for first examples

Task 2

In this task we need to create a C++ function `multiplyFiveMatrices` that takes five `RcppArmadilllo::mat` objects as input, representing matrices A, B, C, D, and E and multiply them in the naive order[10].

```
#include <RcppArmadillo.h>
using namespace arma;
```

[10] Note that an additional function `checkMatrixDimensions()` has been created in order to ensure the dimensions of the matrices the user is trying to multiply are viable.

```cpp
// [[Rcpp::depends(RcppArmadillo)]]


//Function to check if the matrices are of valid dimensions
bool checkMatrixDimensions(const arma::mat& A, const arma::mat& B) {
  int ncolA = A.n_cols;
  int nrowB = B.n_rows;

  //Boolean value
  return (ncolA == nrowB);
}

// Function to multiply five matrices
// Outputs are returned as Rcpp::NumericMatrix objects. Inputs are arma::mat
// [[Rcpp::export]]
Rcpp::NumericMatrix multiplyFiveMatrices(arma::mat A, arma::mat B,
                                         arma::mat C, arma::mat D,
                                         arma::mat E) {

  // Check for dimensions
  if (checkMatrixDimensions(A, B) && checkMatrixDimensions(B, C) &&
      checkMatrixDimensions(C, D) && checkMatrixDimensions(D, E)) {

    // Multiply matrices without explicit parenthesization
    arma::mat result = A*B*C*D*E;

    // Convert Armadillo matrix back to Rcpp::NumericMatrix
    Rcpp::NumericMatrix resultR = Rcpp::wrap(result);

    return resultR;
  } else {
    Rcpp::Rcerr << "Wrong matrix dimensions, please provide correct matrices" << std::endl;
    // Return an empty matrix to indicate failure
    return Rcpp::NumericMatrix();
  }
}
```

Now we can perform a benchmark to compare it to the naive approach using base R and the best example approach using R.[11] This is:

[11] Take into account that we are using `RcppArmaillo` matrices because it is the fastest way to multiply them in C++, not because it is necessarily faster than in R. R matrix multiplication is very fast using `%*%`

```
res <-  microbenchmark::microbenchmark(A %*% B %*% C %*% D %*% E,
                                       A %*% ((B %*% C) %*% (D %*% E)),
                                       multiplyFiveMatrices(A, B, C, D, E),
                                       times = 1000)
```

Table 2: Second Benchmark

expr	min	lq	mean	median	uq	max	neval
A %% *B* %% C %% *D* %% E	68.4	73.3	84.8493	75.45	90.3	3016.3	1000
A %% *((B* %% C) %% *(D* %% E))	8.7	9.3	10.7721	9.60	11.2	78.9	1000
multiplyFiveMatrices(A, B, C, D, E)	30.6	32.9	40.0641	34.10	41.4	2447.7	1000

The results show how even though the Naive multiplication in R is completely outperformed by the C++ implementation with a $\times 2$ boost in performance, this increment is not enough to beat the (nearly) best parenthesization.

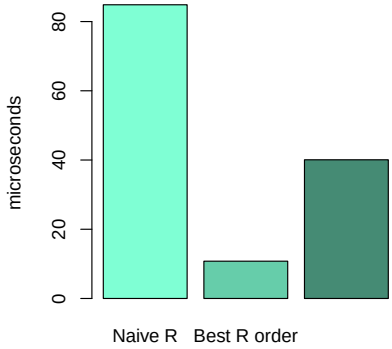

Figure 2: Naive R, nice order and Naive C++ benchmark

Task 3

In task 3 we are asked to implement a function that performs the actual matrix multiplication in a desired order. The matrices should be in a list and the order needs to be a vector.

This task seems easy but it has several complications that quickly become apparent as soon as you stop to think about it. The first thing is how to represent the order.

Let n be the number of matrices you want to multiply. The `order` vector just can't be whatever n-sized vector of integers between $1, \ldots, n$.[12] After discussing with the teacher and some classmates about this fact, the best approach seemed to indicate the multiplicaton order as a *o-indexed vector representing the multiplication itself, not the matrix*.[13] This indexation may cause non-uniqness in the representation but this is no issue in our problem[14].

Now we have decided how to represent the ordering there is one more thing we have to consider. This is the accumulating of matrices in one side of the multiplication[15]. This aspect may seem trivial while mental computation but when abstracting an algorithm represent some serious issues.

The solution we have created for this problem was a history steps vector in order to save the accumulating values each time a multiplication is performed as well as a indicator and and index of whether a multiplication has been performed and when.

[12] This may result in some multiplications that are unfeasible. In the example of task 1, the vector $(1, 5, 2, 3, 4)$.

[13] This is, in the example of task 1, 0 would represent the multiplication between $\mathcal{A}$ and $\mathcal{B}$, 1 would represent the multiplication between $\mathcal{B}$ and $\mathcal{C}$. The parenthesization $(\mathcal{AB})\mathcal{C}(\mathcal{DE})$ will be the vector $(0, 3, 1, 2)$.

[14] For example the $(\mathcal{AB})\mathcal{C}(\mathcal{DE})$ is $(0, 3, 1, 2)$ but also $(0, 1, 3, 2)$

[15] In the $(\mathcal{AB})\mathcal{C}(\mathcal{DE})$, when the 1 multiplication is been performed, the matrix multiplied are not only $\mathcal{B}$ and $\mathcal{C}$. The left side of the multiplication is $\mathcal{AB}$.

The way this functions is simple, for each multiplication in the order vector, we check is the previous or the next multiplication has been performed[16] in order to check for possible accumulations. If it has been performed it checks when has it been performed[17] and goes to that step in the history[18] to recall the specific value of that accumulation. The result of the operation would be the last step in the history when every computation in the order vector is been done. If it has not, then the matrix is taken from the input list. When the operation is performed the result is saved in the history vector and it repeats for the next multiplication.

Note that the integrity of the matrices is not compromised since the dimensions are always preserved. If the order vector is well defined, this algorithm converges always.

The function is the following. It takes a list of matrices and the order vector and multiply them by steps. In each step the left and right hand side of the multiplication are recalled separately.[19] We will be using also `arma::mat`objects because of its quick multipication.

[16] Looking for its Boolean value in the `indicator` vector.
[17] In the `index` vector
[18] In the `steps` vector

[19] Note that the cases for first and last multiplication have been excluded from the conditionals because of index issues.

```
#include <RcppArmadillo.h>
using namespace Rcpp;

// [[Rcpp::depends(RcppArmadillo)]]

// Check matrices dimensions
bool checkMatrixDimensions(const Rcpp::NumericMatrix& A, const Rcpp::NumericMatrix& B) {
    int ncolA = A.ncol();
    int nrowB = B.nrow();
    // Check if dimensions are suitable for multiplication
    return (ncolA == nrowB);
}

// Functions for the well-definition of the order vector
// Checks for duplicates
bool anyDuplicates(IntegerVector vec) {
    std::unordered_set<double> seen;
    for (int i = 0; i < vec.size(); ++i) {
        if (seen.find(vec[i]) != seen.end()) {
            return true;
        } else {
            seen.insert(vec[i]);
        }
    }
    return false;
}
```

```
// Checks dimension
bool checkDimensions(IntegerVector vec, int dims) {
    for (int i = 0; i < vec.size(); ++i) {
        if (vec[i] > dims) {
            return false;
        }
    }
    return true;
}

// [[Rcpp::export]]
arma::mat matrixChainMultiplication(List matrices, IntegerVector order) {
    // Check if the order vector is valid
    if (order.size() != matrices.size() - 1) {
        Rcpp::stop("Invalid order vector size");
    }
    for(int i = 0; i < matrices.size() - 1; i++) {
        if (!checkMatrixDimensions(matrices[i], matrices[i + 1])) {
            Rcpp::stop("Invalid matrices sizes. Check matrix dimensions");
        }
    }
    if (anyDuplicates(order)) {
        Rcpp::stop("Invalid order vector. Check for duplicates");
    }
    if (!checkDimensions(order, matrices.size() - 2)) {
        Rcpp::stop("Invalid order number. Number out of bounds");
    }

    LogicalVector indicator(order.size(), false);
    IntegerVector index(order.size());
    std::vector<arma::mat> steps(order.size());
    int cont = 1;
    arma::mat l, r;

    for (int i = 0; i < order.size(); i++) {
        int curr_order = order[i];
        if (curr_order == 0) {
            // First iteration
            l = as<arma::mat>(matrices[0]);
            if (!indicator[curr_order + 1]) {
                r = as<arma::mat>(matrices[curr_order + 1]);
            } else {
                r = steps[index[curr_order + 1]-1];
```

```
            }
        //last iteration
        } else if (curr_order == order.size() - 1) {
            r = as<arma::mat>(matrices[matrices.size() - 1]);
            if (!indicator[curr_order - 1]) {
                l = as<arma::mat>(matrices[curr_order]);
            } else {
                l = steps[index[curr_order - 1]-1];
            }
        //general logic
        } else {
            if (!indicator[curr_order - 1]) {
                l = as<arma::mat>(matrices[curr_order]);
            } else {
                l = steps[index[curr_order - 1]-1];
            }
            if (!indicator[curr_order + 1]) {
                r = as<arma::mat>(matrices[curr_order + 1]);
            } else {
                r = steps[index[curr_order + 1]-1];
            }
        }
        steps[i] = l * r;
        indicator[curr_order]=true;
        index[curr_order] = cont;
        cont++;
    }

    return steps[order.size() - 1];
}
```

See that this function works as expected

```
matrices <- list(A,B,C,D,E)
order <- as.integer(c(3,2,1,0))

head(matrixChainMultiplication(matrices, order))
```

```
##             [,1]      [,2]
## [1,]   -69.07384  113.0156
## [2,] -1322.68610 -362.0922
## [3,] -1773.21792 -600.5606
## [4,]  -674.11236  448.5666
## [5,]  1174.11051  133.2689
## [6,]   680.00717 -132.3405
```

```
head((A %*% B %*% C %*% D %*% E))
```

```
##              [,1]      [,2]
## [1,]   -69.07384  113.0156
## [2,] -1322.68610 -362.0922
## [3,] -1773.21792 -600.5606
## [4,]  -674.11236  448.5666
## [5,]  1174.11051  133.2689
## [6,]   680.00717 -132.3405
```

We can now compare the best orderig found in task 1 both in R and C++. This is:

```
res1 <-
  microbenchmark::microbenchmark(A %*% ((B %*% C) %*% (D %*% E)),
                                 matrixChainMultiplication(matrices, as.integer(c(1, 3, 2, 0))),
                                 times = 1000)
```

Table 3: Third Benchmark

expr	min	lq	mean	median	uq	max	neval
A %% ((B %% C) %% (D %% E))	8.7	9.7	10.82272	10.2	11.7	36.3	1000
matrixChainMultiplication(matrices, as.integer(c(1, 3, 2, 0)))	16.7	18.7	22.31359	19.8	22.7	1015.6	1000

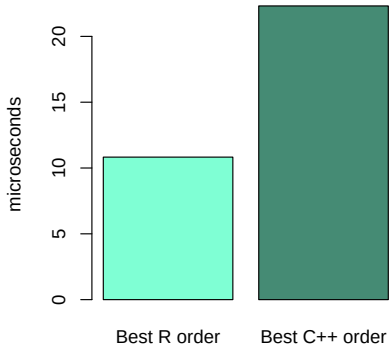

Figure 3: Naive R, nice order and Naive C++ benchmark

This result is pretty surprising. We expected the C++ implementation to be much faster than the R implementation. Assuming the implementation in C++ can't be improved (which isn't true), this illustrates how well R multiplies matrices as a matter of fact. RcppArmadillo is fast but takes its time. This also illustrates how the problem of this assignment is not making the multiplications, but deciding which is the optimal order.

Task 4

Several solutions exist, and in this document we will be exploring the Dynamic Programming approach with memoization[Wikipedia contributors, 2023].

The dynamic programming solution for the matrix chain multiplication problem involves breaking down the problem into smaller sub problems and solving them iteratively. The idea is to compute the minimum number of multiplications required to multiply a sequence of matrices.

This solution builds a table where each entry (i, j) represents the minimum number of multiplications needed to multiply matrices from i to j. The time complexity of this approach is $O(n^3)$, where n is the number of matrices in the sequence. By applying memoization[20], we can reduce the time complexity to $O(2^n)$ by avoiding redundant computations.

[20] Memoization is a technique used to optimize recursive algorithms by storing the results of expensive function calls and reusing them when the same inputs occur again.

Exponential algorithms always seem as a *wannabe* in computer science. There is a solution in the scientific literature from [Hu and Shirig] that allows for a $O(nlogn)$ algorithm in order to determine the best multiplication order given a matrix chain. This is achieved by the identification of the matrix chain multiplication problem with the partition of convex polygons problem with costs. This solution is very interesting but it has not been implemented because of its theoretical complexity.

However the solution given in this document is very understandable. The following function is iterative. For each (double) iteration, the cost in terms of computations is updated if the new cost is better than the olds one. In order not to get a matrix representation, a recursive algorithm has been used in order to reconstruct the parenthesization with the help of AI.

The easiest way[21] to make the function is to give it as input a vector of the chain representation of the dimensions of the matrices[22]. Note that a function to automatically generate this vector is given.

[21] memory-wise and more general for other problems

[22] This is without the duplicates resulting from the column-row equities and the values in order of multiplication

The output of the function consists of the matrix with the costs and the reconstruction of the parenthesization.

```
#include <RcppArmadillo.h>
using namespace Rcpp;
using namespace arma;

// Function to compute minimum number of scalar multiplications and optimal parenthesization
// dims: dimensions of matrices
// Returns a list containing the minimum cost matrix and the optimal split points matrix
// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]
List matrixChainMultDP(arma::ivec dims) {
  int n = dims.size() - 1;
  arma::imat m(n, n, arma::fill::zeros); // Minimum number of scalar multiplications
  arma::imat s(n, n, arma::fill::zeros); // Index of the subsequence split achieving minimal cost

  // Dynamic programming approach to compute minimum number of scalar multiplications
  for (int len = 2; len <= n; len++) {
    for (int i = 0; i < n - len + 1; i++) {
      int j = i + len - 1;
```

```
      m(i, j) = std::numeric_limits<int>::max();
      for (int k = i; k < j; k++) {
        int cost = m(i, k) + m(k + 1, j) + dims(i) * dims(k + 1) * dims(j + 1);
        if (cost < m(i, j)) {
          m(i, j) = cost;
          s(i, j) = k;
        }
      }
    }
  }

  // Function to recursively construct optimal parenthesization
  std::function<std::string(int, int)> constructOptimalParenthesization;
  constructOptimalParenthesization = [&](int i, int j) -> std::string {
    if (i == j) {
      return "M" + std::to_string(i + 1);
    } else {
      int k = s(i, j);
      std::string left = "(" + constructOptimalParenthesization(i, k) + ")";
      std::string right = "(" + constructOptimalParenthesization(k + 1, j) + ")";
      return left + "*" + right;
    }
  };

  std::string optimalParenthesization = constructOptimalParenthesization(0, n - 1);

  return List::create(Named("minCostMatrix", m),
                      Named("optimalParenthesization", optimalParenthesization));
}

// Function to create array of dimensions from list of matrices
// matrices: list of matrices
// Returns array of dimensions
// [[Rcpp::export]]
arma::ivec createDimsVector(List matrices) {
  int n = matrices.size();
  arma::ivec dims(n + 1);

  for (int i = 0; i < n; i++) {
    NumericMatrix mat = matrices[i];
    dims[i] = mat.nrow();
  }
  dims[n] = as<NumericMatrix>(matrices[n - 1]).ncol();
```

```
  return dims;
}
```

Lets try the function for our task 1 examples:

```
dims <- createDimsVector(matrices)
result <- matrixChainMultDP(dims)
print(result$optimalParenthesization)
```

```
## [1] "(M1)*((M2)*((M3)*((M4)*(M5))))"
```

We could also try it with some computational expensive matrices.
The optimal order for a 10 big matrices chain is:

```
# Create a chain of 10 matrices with extreme dimensions
matrices_big <- list()

matrices_big[[1]] <- matrix(nrow = 1000, ncol = 4)
matrices_big[[2]] <- matrix(nrow = 4, ncol = 1340)
matrices_big[[3]] <- matrix(nrow = 1340, ncol = 400)
matrices_big[[4]] <- matrix(nrow = 400, ncol = 400)
matrices_big[[5]] <- matrix(nrow = 400, ncol = 23)
matrices_big[[6]] <- matrix(nrow = 23, ncol = 1000)
matrices_big[[7]] <- matrix(nrow = 1000, ncol = 7)
matrices_big[[8]] <- matrix(nrow = 7, ncol = 6)
matrices_big[[9]] <- matrix(nrow = 6, ncol = 30)
matrices_big[[10]] <- matrix(nrow = 30, ncol = 2)

dims_big <- createDimsVector(matrices_big)
result_big <- matrixChainMultDP(dims_big)
print(result_big$minCostMatrix)
```

```
##       [,1]    [,2]    [,3]      [,4]     [,5]     [,6]    [,7]    [,8]    [,9]
##  [1,]    0 5360000 3744000   4384000  2912800  6912800 2968800 2964968 3061688
##  [2,]    0       0 2144000   2784000  2820800  2912800 2940800 2940968 2941688
##  [3,]    0       0       0 214400000 16008000 46828000 5097400 4393166 4634366
##  [4,]    0       0       0         0  3680000 12880000 1345400 1177166 1249166
##  [5,]    0       0       0         0        0  9200000  225400  217166  289166
##  [6,]    0       0       0         0        0        0  161000  161966  166106
##  [7,]    0       0       0         0        0        0       0   42000  211260
##  [8,]    0       0       0         0        0        0       0       0    1260
##  [9,]    0       0       0         0        0        0       0       0       0
## [10,]    0       0       0         0        0        0       0       0       0
##         [,10]
##  [1,] 1489564
##  [2,] 1481564
```

```
##  [3,] 1470844
##  [4,]  398844
##  [5,]   78844
##  [6,]   60444
##  [7,]   14444
##  [8,]     444
##  [9,]     360
## [10,]       0
```

```
print(result_big$optimalParenthesization)
```

```
## [1] "(M1)*((M2)*((M3)*((M4)*((M5)*((M6)*((M7)*((M8)*((M9)*(M10))))))))))"
```

We can compare this results with the naive order reducing the times the `microbenchmark`function performs the operations.

```
res2 <-  microbenchmark::microbenchmark(
  matrices_big[[1]] %*% matrices_big[[2]] %*% matrices_big[[3]] %*% matrices_big[[4]]
  %*% matrices_big[[5]] %*% matrices_big[[6]] %*% matrices_big[[7]] %*%
  matrices_big[[8]] %*% matrices_big[[9]] %*% matrices_big[[10]],
  matrices_big[[1]] %*% (matrices_big[[2]] %*% (matrices_big[[3]] %*%
  ( matrices_big[[4]] %*% ( matrices_big[[5]] %*% (matrices_big[[6]] %*%
  ( matrices_big[[7]] %*% (matrices_big[[8]] %*% ( matrices_big[[9]] %*%
  matrices_big[[10]])))))))),
    matrixChainMultiplication(matrices_big, as.integer(c(2, 8, 7, 6, 5, 3, 1, 0))),
    times = 10
  )
```

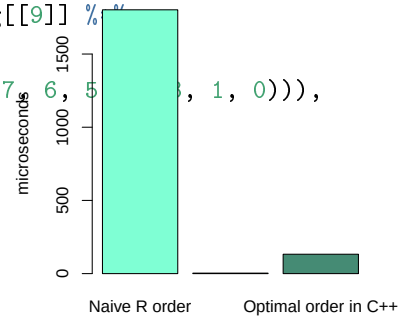

Figure 4: Naive R, nice order and Naive C++ benchmark

This results verify the results of the previous section.

Further work

Having solved all four of the tasks required in this assignment, some further topics may arise.

Talking about programming, a function that translates automatically the output of the `matrixChainMultDP()` function to a 0-indexed vector as the input of the `matrixChainMultiplication()` function seem like the next step. This has not been done because of deadline issues. This implementation will help in order to communicate both functions in a nested way. It may be interesting if the user wants to create a package for example.

Talking about performance, two main topics point out. The first of them is to implement the $O(nlogn)$ solution and compare it to the exponential one in order to show its power. As we have commented, this solution requires deep mathematical concepts.

On the other hand, the Dynamic Programming procedure has a 3-nested loop block inside it. This nested loops are always object of parallelization. Parallel programming may be useful to *mitigate* the effects of exponential behavior by using several cores in the computation. This could be achieved using C++ libraries such as `RcppParallel`and `OpenMP`. Also a version in R using parallel programming may be interesting to compare the results. Even though the parallel programming was inside the scope of this course, deadlines are often hard to meet and even though we tried a few examples, C++ debugging is difficult and we couldn't make it to work. It will be surely explored in future works.

References

Dirk Eddelbuettel and James Joseph Balamuta. Extending R with C++: A Brief Introduction to Rcpp. *The American Statistician*, 72(1): 28–36, 2018. DOI: 10.1080/00031305.2017.1375990.

Dirk Eddelbuettel and Conrad Sanderson. Rcpparmadillo: Accelerating r with high-performance c++ linear algebra. *Computational Statistics and Data Analysis*, 71:1054–1063, March 2014. DOI: 10.1016/j.csda.2013.02.005.

T. C. Hu and ICI. 'I'. Shirig. Computation of Matrix Chain Products, Part I, Part II. `http://i.stanford.edu/pub/cstr/reports/cs/tr/81/875/CS-TR-81-875.pdf`. [Accessed 17-03-2024].

Olaf Mersmann. *microbenchmark: Accurate Timing Functions*, 2023. URL `https://CRAN.R-project.org/package=microbenchmark`. R package version 1.4.10.

Wikipedia contributors. Matrix chain multiplication — Wikipedia, the free encyclopedia, 2023. URL `https://en.wikipedia.org/w/index.php?title=Matrix_chain_multiplication&oldid=1192037372`. [Online; accessed 17-March-2024].