Lab 3 Solving the matrix search problem

C S 372 Data Structures and Algorithms

September 4, 2019

In this lab, you will design and implement algorithms to solve the matrix search problem—finding the maxima of each row in the matrix. The problem is disguised as different problems in many applications. They include breaking paragraphs into lines, sequence alignment, RNA secondary structure prediction, and image thresholding.

1 The matrix search problem

The matrix search problem finds the maxima of each row in a matrix.

Input: an $m \times n$ matrix

$$\begin{bmatrix} 0 & -4 & -1 & 2.5 & \color{red}{4} \\ -3 & \color{red}{8} & -10 & 2 & 7 \\ -4 & -3 & \color{red}{-1} & -100 & -5.5 \\ 0 & 2 & 0.3 & -3 & \color{red}{2.5} \\ 1 & \color{red}{3} & 1 & 2 & 0 \\ -8 & 9 & \color{red}{10} & 2 & 5 \end{bmatrix}$$

Output: the maximum element of each row, marked in red.

2 A brute-force iterative solution

This algorithm searches for the maxima row by row. On each row, it linearly scans each element to find the maximum. The runtime is apparently $O(mn)$. Implement this algorithm using a C++ program. Please define your function as follows:

```
template<typename T>
vector<T> find_row_maxima_itr(const matrix<T> & m)
{
  // Your code here
}

//[[Rcpp::export]]
vector<double> row_maxima_itr
  (const vector<double> & v, size_t nrow, size_t ncol)
{
  matrix<double> mat(v, nrow, ncol);
  return find_row_maxima_itr(mat);
}
```

The input matrix is represented in the vector `v` column by column (not row by row). This choice is made so that the R `matrix` class can be passed to the C++ `vector` class without any data type conversion, saving time. Internally in R, a matrix is already stored as a column-major vector.

This function uses a C++ template matrix class given in the skeleton code file `MatrixSearch-Skeleton.cpp`.

3 Divide-and-conquer monotonic matrices

Let $j(i)$ define the smallest column index to the maximum element in row i. A matrix is monotone if for any $i_1 < i_2$, $j(i_1) \leq j(i_2)$. In the following example

$$\begin{bmatrix} 0 & 4 & -1 & 2.5 & -4 \\ -3 & 8 & -10 & 2 & 7 \\ -4 & -3 & -1 & -100 & -5.5 \\ 0 & 2 & 0.3 & -3 & 2.5 \\ 1 & 0 & 1 & 2 & 3 \\ -8 & 9 & 2 & 5 & 10 \end{bmatrix}$$

we have the following column positions of the row maxima

$$j(0) = 1 \leq j(1) = 1 \leq j(2) = 2 \leq j(3) = 4 \leq j(4) = 4 \leq j(5) = 4$$

Please note that the column indices of the row maxima are non-decreasing (1, 1, 2, 4, 4, 4), but the row maximum values are not necessarily increasing. In this example, they are 4, 8, -1, 2.5, 3, 10—the row maximum values can go up and down.

For a monotonic matrix, we can perform divide-and-conquer on the rows to solve the matrix search problem much faster than the brute-force iterative solution. Please design a divide-and-conquer strategy to achieve a runtime of $O(n \lg m)$, using the following template

```
template<typename T>
vector<T> find_row_maxima(const matrix<T> & m)
{
  // Your divide-and-conquer code here
}

//[[Rcpp::export]]
vector<double> row_maxima
  (const vector<double> & v, size_t nrow, size_t ncol)
{
  matrix<double> mat(v, nrow, ncol);
  return find_row_maxima(mat);
}
```

4 Test the two functions

Develop a C/C++ test function to include five examples to check your functions. If a function passes the tests, minimal output should be displayed on the screen to indicate success; otherwise, point out which example the function failed. This test function should take a function parameter so that it can test both of your functions, defined as follows:

```
bool test_row_maxima
(vector<double> (*rmfun) (const vector<double> & v,
                          size_t nrow, size_t ncol))
{
  // Example 1
  bool passed = true;
  /* Monotonic matrix example 1:
   0,  4, -1, 2.5, -4,
  -3, 8, -10, 2, 7,
  -4, -3, -1, -100, -5.5,
  0, 2, 0.3, -3, 2.5,
  1, 0, 1, 2, 3,
  -8, 9, 2, 5, 10};
  */
  // x is column major vectorization of the matrix
  double x[] = {  0,  -3,   -4,   0, 1, -8,
                  4,   8,   -3,   2, 0,  9,
                  -1, -10,   -1, 0.3, 1,  2,
                  2.5,   2, -100,  -3, 2,  5,
                  -4,   7, -5.5, 2.5, 3, 10};

  vector<double> v(x, x+30);

  double rmax_truth[] = {4, 8, -1, 2.5, 3, 10};

  if(rmfun(v, 6, 5) != vector<double>(rmax_truth, rmax_truth+6)) {
    cout << ”ERROR: failed test 1!” << endl;
    passed = false;
  }

  // Example 2

  ...

  // Example 5

  return passed;

}

//[[Rcpp::export]]
bool testall()
{
  bool passed = true;
  if(!test_row_maxima(row_maxima_itr)) {
    cout << ”ERROR: row_maxima_itr() failed some test!” << endl;
    passed = false;
  }

  if(!test_row_maxima(row_maxima)) {
    cout << "ERROR: row_maxima() failed some test!" << endl;
    passed = false;
  }

  if(passed) {
    cout << "All tests passed. Congratulations!" << endl;
  }
  return passed;
}
```

Your C++ program must include a main() function that calls the `testall()` function. When the program is compiled by a C++ compiler it generates a binary executable file that will run when invoked from the command line.

5 Visualize the runtime of the two methods

Develop R code inside your C++ source code file to generate runtime report on the two functions you developed.

You should use the following R function to generate any sized monotonic matrix:

`random.monotone.matrix(nrow, ncol)`

which is already provided in the skeleton code. In the runtime evaluation you can focus on testing square matrices ($m = n$).

You will generate one plot containing four curves as shown in Figure 1. It shows both empirical runtime and estimated theoretical runtime. Your code will estimate the coefficients c_1 and c_2 so that you can plot the theoretical curves fitted to the runtime data.

You need to call the `testall()` function first. If the test failed, visualization should not proceed and you must fix the code first.

6 Submission

Write a lab report to describe your lab work done in the following sections:

1. Introduction (define the background, motivation, and the problem),

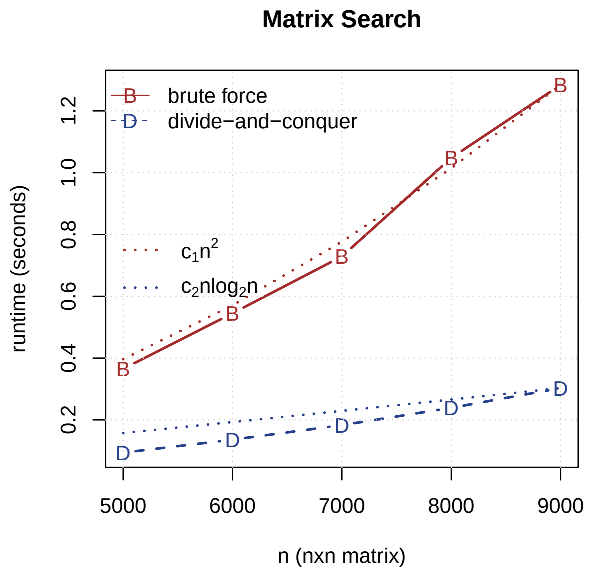

Figure 1: Monotonic matrix search runtime. You are required to show the empirical and theoretical runtimes of the brute-force iterative and the divide-and-conquer recursive solutions.

2. Methods (provide the solutions),
3. Results
   (a) show numbers, tables, or figures.
   (b) report the estimated c_1 and c_2 for the runtime of the two functions on your computer.
4. Discussions (general implications and issues),
5. Conclusions (summarize the lab and point to a future direction).

Submit the source code files and your lab report online.