Advance Programming - Homework 1

Alessandro Castelli ID: 12246581

October 22, 2023

Contents

1 Introduction **3**

2 Matrix multiplication **3**
2.1 My implementation 3
2.2 Results 4

3 Sieve of Eratosthenes **4**
3.1 My implementation 4
3.2 Result 5

4 BubbleSort and BucketSort **5**
4.1 BubbleSort implementation 5
4.2 BucketSort implementation 6
4.3 BubbleSort Result 7
4.4 BucketSort Result 7

1 Introduction

In this document, you will find the report for the exercises included in the "Homework1" folder. The code can be found on the course's Moodle page.

2 Matrix multiplication

The first exercise requires implementing matrix multiplication. To perform the multiplication between two matrices, it is necessary that the number of columns in the first matrix is equal to the number of rows in the second matrix. If this condition is not met, the multiplication is not feasible. The values of the resulting matrix are calculated following this formula:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

2.1 My implementation

I have created a Matrix class as follows:

Listing 1: Matrix Class

```
class Matrix{
    private:
    vector<vector<int>> matrix;

    public:
        Matrix(int rows, int columns);
        void setElement(int row, int column, int value);
        void show();
        int getNumberRows();
        int getNumberColumns();
        Matrix multiplication(Matrix m);
};
```

I have created several functions, but the one that performs the multiplication is: `Matrix multiplication(Matrix m)`.

Listing 2: Matrix multiplication

```
Matrix Matrix::multiplication(Matrix m) {
    int nRow1 = this->getNumberRows();
    int nCol1 = this->getNumberColumns();
    int nRow2 = m.getNumberRows();
    int nCol2 = m.getNumberColumns();

    if (nCol1 != nRow2){
        cout<<"\n"<<"Unable to perform multiplication because
           the matrices have unsuitable dimensions."<<endl;
        return *this;
    } else {
        Matrix result(nRow1,nCol2);
        for(int i = 0; i < nRow1; ++i)
            for(int j = 0; j < nCol2; ++j)
```

```
14                  for (int k = 0; k < nCol1; ++k)
15                      result.matrix[i][j] += this->matrix[i][k] *
                            m.matrix[k][j];
16          return result;
17      }
18  }
```

If the number of columns in the first matrix is different from the number of rows in the second matrix, then I do not perform the multiplication and print an error message; otherwise, I carry out the multiplication. I have created three `for` loops that allow me to perform column-by-row multiplication. Afterward, I save the result in a new matrix, which is also the one to be returned in the end.

2.2 Results

To test my code, I have created two matrices for which I can set the dimensions, and they contain values ranging from 0 to 100.

Here are some time results that give an idea of the time required to perform matrix multiplication in the traditional way:

- Dimension $MatrixA = 100 \times 200$, Dimension $MatrixB = 200 \times 100$, $Time = 0.0228102s$
- Dimension $MatrixA = 1000 \times 2000$, Dimension $MatrixB = 2000 \times 1000$, $Time = 51.1343s$
- Dimension $MatrixA = 2000 \times 4000$, Dimension $MatrixB = 4000 \times 2000$, $Time = 364.741s$

Matrix multiplication algorithm has an asymptotic complexity of $O(n^3)$ because it is necessary to execute three nested `for` loops

3 Sieve of Eratosthenes

The second exercise necessitates the implementation of the Sieve of Eratosthenes, an algorithm designed to compute prime numbers up to a specified upper threshold.

3.1 My implementation

I have created a `EratosthenesSeries` algorithm as follows:

Listing 3: EratosthenesSeries

```
1  EratosthenesSeries::EratosthenesSeries(int max) {
2      this->maximum=max;
3      primes.resize(max);
4      for (int i=0; i<=max; i++) {
5          primes[i]= true;
6      }
7      primes[0] = false; primes[1]=false;
8
9      for (int p = 2; p * p <= maximum; p++) {
10         if (primes[p]) {
11             for (int i = p * p; i <= maximum; i += p) {
```

```
                primes[i] = false;
            }
        }
    }
}
```

As can be seen from 3, the algorithm uses a vector as large as the upper limit and initializes it with boolean values. Then, starting from the integer two, it begins to set all multiples of two to false, then all multiples of three, and so on until i remains less than the maximum threshold.

3.2 Result

I have timed the execution of my algorithm by clocking it on different inputs and these are the results I have obtained:

- $Inputsize : 1000000, Time = 0.369873s$
- $Inputsize : 10000000, Time = 2.046s$
- $Inputsize : 100000000, Time = 20.9225s$
- $Inputsize : 1000000000, Time = 437.616s$

The Sieve of Eratosthenes has an asymptotic space complexity of $O(n)$ as it uses a vector that is the same size as the input, and it has a time complexity of $O(n \cdot log(log(n)))$.

4 BubbleSort and BucketSort

The third and fourth exercises require the implementation of two sorting algorithms: BubbleSort and BucketSort. These algorithms take an integer vector as input and return a vector of the same size but with values sorted in ascending order.

4.1 BubbleSort implementation

I have created a `bubbleS` algorithm as follows:

Listing 4: BubbleSort

```
void BubbleSort::bubbleS() {
    int size = data.size();

    for (int i = 0; i < size-1; i++) {
        for (int j = 0; j < size - i -1 ; j++) {
            if (data[j]>data[j+1]){
                int temp = data[j];
                data[j]=data[j+1];
                data[j+1]=temp;
            }
        }
    }
}
```

As can be seen in 4, what the algorithm does is compare the two adjacent numbers and swap them if they are in the wrong order. After each iteration, the largest element among the unsorted elements is placed at the end.

4.2 BucketSort implementation

I have created a `bucketS` algorithm as follows:

Listing 5: BucketSort

```
void BucketSort::bucketS() {
    int minValue = data[0];
    int maxValue = data[0];

    //Find Maximum and Minimum
    for (int i = 1; i < this->data.size(); i++)
    {
        if (data[i] > maxValue)
            maxValue = data[i];
        if (data[i] < minValue)
            minValue = data[i];
    }
    int bucketLength = maxValue - minValue + 1;
    vector<vector<int>> bucket(bucketLength);
    for (int i = 0; i < bucketLength; i++)
    {
        bucket[i] = vector<int>();
    }
    for (int i = 0; i < this->data.size(); i++)
    {
        bucket[data[i] - minValue].push_back(data[i]); //This
           command is distributing the elements of the ‘data‘
           vector into the corresponding "buckets" based on
           their value minus ‘minValue‘.
    }
    int k = 0;
    for (int i = 0; i < bucketLength; i++)
    {
        int bucketSize = bucket[i].size();
        if (bucketSize > 0)
        {
            for (int j = 0; j < bucketSize; j++)
            {
                data[k] = bucket[i][j];
                k++;
            }
        }
    }
}
```

As can be seen in 5, The algorithm works as follows:

1. **Find the minimum and maximum value**: This step is essential to determine the length of the bucket vector
2. **Initialize the buckets**: Create a vector of vectors of the length calculated in step 1. Each element of the 2D vector is an empty vector
3. **Distribute the elements into the buckets**: For each element in the array, calculate the bucket index as `element - minimum value` and add the element to the corresponding bucket
4. **Collect the elements from the buckets back to the original array**: Iterate over each bucket and each element within each bucket, and copy the elements in the buckets back into the original array one by one.

4.3 BubbleSort Result

I have timed the execution of my algorithm by clocking it on different inputs and these are the results I have obtained:

- $Inputsize = 10000, Time = 0.628202s$
- $Inputsize = 100000, Time = 58.2653s$
- $Inputsize = 200000, Time = 222.019s$

This algorithm has a time complexity of $(O(n^2))$ as there are two nested loops.

4.4 BucketSort Result

I have timed the execution of my algorithm by clocking it on different inputs and these are the results I have obtained:

- $Inputsize = 10000, Time = 0.0009811ss$
- $Inputsize = 100000, Time = 0.0034783ss$
- $Inputsize = 200000, Time = 0.0066785s$

The Bucket Sort algorithm has a time and space complexity of $O(n + k)$ in the average and best case, where `n` is the number of elements and `k` is the number of buckets. However, in the worst case, the time complexity can become $O(n^2)$ if all elements end up in the same bucket.