Fhelicia Selas Selestin
12th April 2018

NETVIRTA CODING CHALLENGE

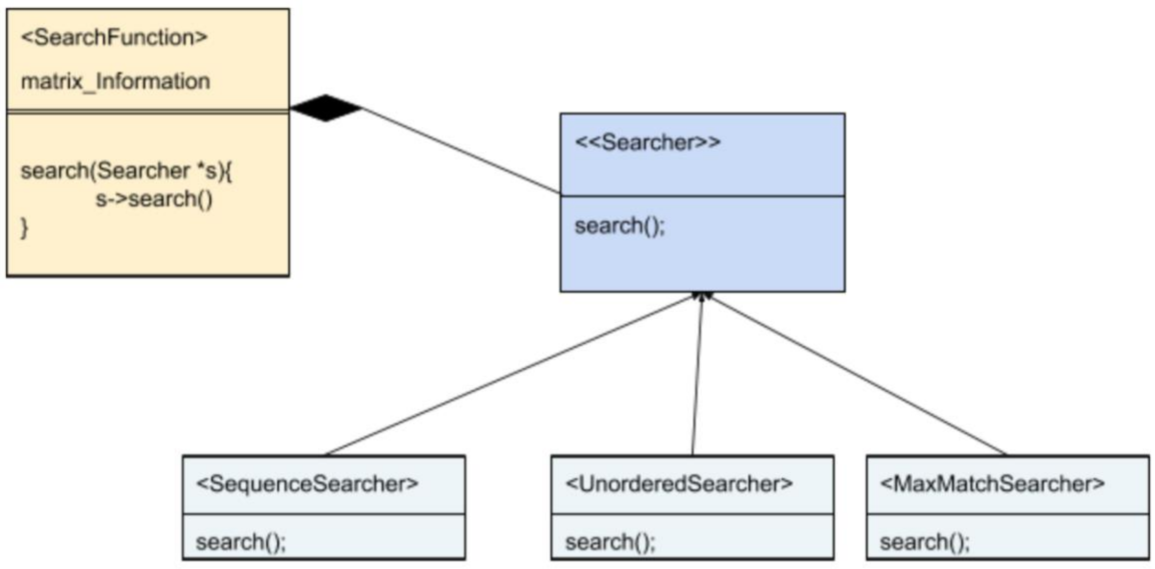
Problem Description

Given a matrix stored in encrypted format, decrypt it and search for numbers in the matrix. The various search functions to be performed are:

1. Searching a Sequence: Print rows which contain the sequence
2. Searching for numbers: Print rows which have all the numbers (including repetitions)
3. Searching Maximum matching row: Print row which has maximum matches.

Design

The Program is designed as follows:



The Searcher class is an abstract class containing the abstract pure virtual function ‘search’

The classes SequenceSearcher,UnorderedSearcher and MaxMatchSearcher inherit the Searcher class and implement the ‘search’ method.

The class ‘SearchFunction’ holds the private matrix information and also has a search method with a Searcher pointer as a method argument. It calls the ‘search’ method of the class the current Searcher pointer is pointing to. (Dynamic binding)

To create a new Search function, a new class inheriting the Searcher class and implementing the search function needs to be created.

Based on the user input, the object corresponding to the Search class is passed to the search method of the ‘SearchFunction’ class.

Implementation

The encrypted file is passed as a command line argument.

The file is decrypted and matrix information is stored in the SearchFunction class. The matrix information is stored in two ways.

1. The matrix itself, a 2D array implemented as <vector <vector <int > > >

2. A Map, elementCountMap, containing every number in the matrix as a key, and value as a vector of pairs of row numbers and the count of the number of times it appears in the row. (unordered_map<int, vector < pair <int,int> >)

Eq:
1 2 3
1 1 1
3 1 4

The elementCountMap is as follows:

Key	Value
1	[<1,1> , <2,3> , <3,1>]
2	[<1,1>]
3	[<1,1>, <3,1>]
4	[<3,1>]

So, the first entry means, the number 1 appears 1 time in row 1 , 3 times in row 2, 1 time in row 3.

The above map is precomputed once, the first time the program runs. And multiple searches are done on the same precomputed data.

The Search Functions are implemented as follows:

Consider the matrix to be a n x c matrix. The input search string is of length 'm'.

Unordered Searcher:

1. A count array equal to the size of the number of rows in the matrix is created.

   count[n]

2. The input numbers to be searched are sorted. O(m log m)

3. Since, the input is sorted, the number of repetitions of a particular number is computed by traversing linearly O(m) and stored in 'ct'.

4. The vector of pairs for the number is obtained

5. For every pair, it is checked if row has at least 'ct' occurrences of the number. If so, the count of that row number is incremented by ct.

   This is done once per number and not for all the repetitions of that number.

6. Finally, the count array is checked to find the rows which have value equal to the input numbers' length

7. The rows matching the condition in 5 are returned as the result.

Sequence Searcher:

An original copy of the input sequence is stored.

Steps 1 to 6 of the above(Unordered Searcher) are repeated.

7. A sequence search of the input is performed only on the rows obtained from step 6 and rows containing the sequence are returned as result.

Max Match Searcher

Steps 1 – 4 of the unordered searcher are repeated.

5. For ever pair,
   a. if the row has value more than the number of repetitions(ct), the count[row_num] is incremented by ct.
   b. If row has value less than the number of repetitions, then that value is added to the count array

6. Finally, the count array is traversed to find the first row with the maximum value of occurrences.

7. The row with the maximum value is returned as output.

Running the code:

Compilation:

```
cd code
make clean
make
```

Run:

```
./MatrixSearcher  ../inputfile/encrypted.txt
```

Utilities

Encrypted Input:

The file encrypt_text.cc takes an input plain text file containing the matrix as input and produces an encrypted file in the SearchMatrix/inputfiles folder (‘encrypted.txt)

A simple XOR encryption is done to every row in the matrix with the key ‘N’.
This encrypted file can be used as input for the Searching Program.

Running the code:

Compilation:	`g++ -g -o encrypt_text encrypt_text.cc`
Run:	`./encrypt_text inputfilename`

Create Matrix

Simple class which takes as input the dimensions of the 2 D matrix and creates a matrix file.

Running the code:

Compilation:	`g++ -g -o  createMatrix createMatrix.cc`
Run:	`./createMatrix`

Tests

Simple Unit tests covering cases: Number found in matrix, Number not found in matrix, Number found in first row, Number found in last row, and finding rows with exact number repetitions have been written for all the search functions

Running the code:

Compilation:

```
cd tests
make clean
make
```

Run: `./TestSearcher`

Benchmark

High resolution clock is used to record the start and end times of the search functions. The micro seconds required for various dimensions of the matrix are recorded by calculating the average of 1000 runs of a search.

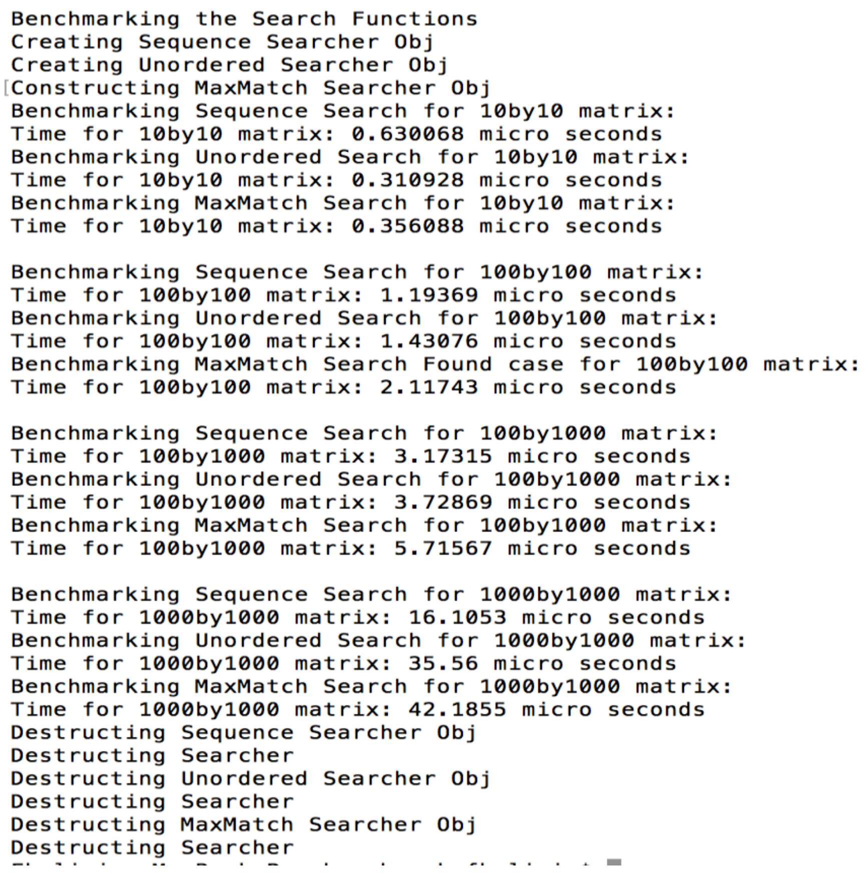
Running the code:

Compilation:

```
cd benchmark
make clean
make
```

Run: `./BenchMarker`

A screen shot of one run of the Benchmark code can be found below:



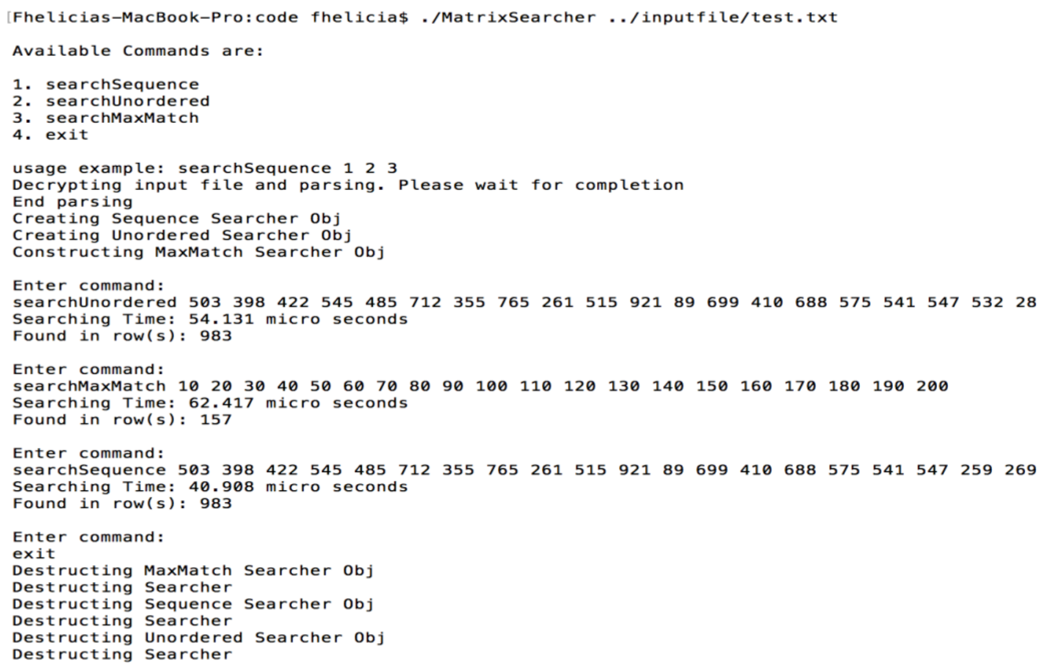
```
Benchmarking the Search Functions
Creating Sequence Searcher Obj
Creating Unordered Searcher Obj
Constructing MaxMatch Searcher Obj
Benchmarking Sequence Search for 10by10 matrix:
Time for 10by10 matrix: 0.630068 micro seconds
Benchmarking Unordered Search for 10by10 matrix:
Time for 10by10 matrix: 0.310928 micro seconds
Benchmarking MaxMatch Search for 10by10 matrix:
Time for 10by10 matrix: 0.356088 micro seconds

Benchmarking Sequence Search for 100by100 matrix:
Time for 100by100 matrix: 1.19369 micro seconds
Benchmarking Unordered Search for 100by100 matrix:
Time for 100by100 matrix: 1.43076 micro seconds
Benchmarking MaxMatch Search Found case for 100by100 matrix:
Time for 100by100 matrix: 2.11743 micro seconds

Benchmarking Sequence Search for 100by1000 matrix:
Time for 100by1000 matrix: 3.17315 micro seconds
Benchmarking Unordered Search for 100by1000 matrix:
Time for 100by1000 matrix: 3.72869 micro seconds
Benchmarking MaxMatch Search for 100by1000 matrix:
Time for 100by1000 matrix: 5.71567 micro seconds

Benchmarking Sequence Search for 1000by1000 matrix:
Time for 1000by1000 matrix: 16.1053 micro seconds
Benchmarking Unordered Search for 1000by1000 matrix:
Time for 1000by1000 matrix: 35.56 micro seconds
Benchmarking MaxMatch Search for 1000by1000 matrix:
Time for 1000by1000 matrix: 42.1855 micro seconds
Destructing Sequence Searcher Obj
Destructing Searcher
Destructing Unordered Searcher Obj
Destructing Searcher
Destructing MaxMatch Searcher Obj
Destructing Searcher
```

Sample Input/Output



```
Fhelicias-MacBook-Pro:code fhelicia$ ./MatrixSearcher ../inputfile/test.txt

Available Commands are:

1. searchSequence
2. searchUnordered
3. searchMaxMatch
4. exit

usage example: searchSequence 1 2 3
Decrypting input file and parsing. Please wait for completion
End parsing
Creating Sequence Searcher Obj
Creating Unordered Searcher Obj
Constructing MaxMatch Searcher Obj

Enter command:
searchUnordered 503 398 422 545 485 712 355 765 261 515 921 89 699 410 688 575 541 547 532 28
Searching Time: 54.131 micro seconds
Found in row(s): 983

Enter command:
searchMaxMatch 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200
Searching Time: 62.417 micro seconds
Found in row(s): 157

Enter command:
searchSequence 503 398 422 545 485 712 355 765 261 515 921 89 699 410 688 575 541 547 259 269
Searching Time: 40.908 micro seconds
Found in row(s): 983

Enter command:
exit
Destructing MaxMatch Searcher Obj
Destructing Searcher
Destructing Sequence Searcher Obj
Destructing Searcher
Destructing Unordered Searcher Obj
Destructing Searcher
```