



EE441- Programming Assignment 3

Due Date: 13.01.2023, 23:55

For your questions: Ferhat Gölbol, Utkucan Doğan – {ferhatg,utkucan}@metu.edu.tr

This assignment consists of three parts. You are going to create a separate Code::Blocks project for both parts and a pdf file for the last part. Don't forget to write comments to your code as they are also graded.

Consider the recursive matrix determinant calculation with cofactor method. For a sparse matrix with few off-diagonal entries, many cofactor matrices are equal to each other. Naïve implementation recalculates the determinant repeatedly for these cofactor matrices. This algorithm has complexity $O(N!)$, which grows very fast with the matrix size.

In this assignment, you will implement a memoizing determinant calculation function. In this approach, the function keeps memory of previous function arguments and results. When the function is called, it checks the storage. If the argument exists, then retrieves the result from the storage. If it does not exist, calculates the result and stores it in the storage. A pseudocode for this approach is as follows:

```
function determinant of Matrix M
    if M exists in storage
        return result corresponding to M from storage
    else
        calculate result recursively using cofactor method
        store (M, result) in storage
        return result
```

You will implement a Matrix class to be used with both parts. **Do not use template size for the matrix class**, as it would make storage implementation more complex. You can assume that matrix is square and maximum matrix size, N , is 20x20, thus you may declare a 20x20 static array and use $N \times N$ portion of it. Matrix elements are of integer type. Your matrix class must have at least the following member functions:

- Default constructor that initializes the matrix as a 0x0 matrix, a constructor that takes size N and initializes the matrix as $N \times N$ identity matrix and a copy assignment operator
- A getter and a setter function to get and set an element for a given pair of (row,column)
- Custom-defined comparison operators, `operator==` and `operator<` (for use in part 1, see Part 1 for details)
- A hash function (for use in part 2, see Part 2 for details)

Part 1 – Memoization with Binary Search Tree [40 points]

In this part, you will implement a binary search tree and use it as the storage element for the determinant calculation function. A binary tree is a tree in which the maximum degree of any node is 2. A binary search tree is a binary tree in which data values in the left sub-tree of every node are “less than” the data value in the node and those in the right sub-tree are “greater”. For this assignment, the tree does not need to be balanced. A BST node is defined as follows:

```
class BST_Node
{
    public :
        BST_Node* left;
        BST_Node* right;
        // Determinant of key matrix is value
        Matrix key;
        long value;
};
```

Ordering of the nodes will be done with respect to key; that is, a `BST_Node, A`, is smaller than another one, `B`, if `A.key < B.key`. For matrix comparison operators,

- Matrix `A` is equal to `B` if
 - `A.N == B.N`, where `N` is matrix size, and
 - `A.element(i,j) == B.element(i,j)` for all $0 \leq i,j < N$.
- Matrix `A` is smaller than `B` if
 - `A.N < B.N`, or
 - `A.N == B.N` and `A.element(i,j) < B.element(i,j)` where (i,j) is the location of the first non-equal element of both matrices. (Compare elements at the first column of first row. If they are equal, check second column of first row, if still equal check third column of first row and so on. If all elements in the first row are equal, continue comparison with the second row, then third row and so on.)

Implement these comparison operators. Implement the following functions for a binary search tree:

- `bool key_exists(Matrix A)`, that returns `true` if the tree has a `BST_Node` with `key == A` and `false` otherwise. **Hint:** The BST is sorted with respect to key fields. Left sub-tree of any node has smaller keys than root's key and right sub-tree has larger keys. Thus, your function does not need to traverse the whole tree.
- `long search(Matrix A)`, that returns the `value` field of the node whose `key == A`.
- `void insert(Matrix A, long detA)`, that creates a `BST_Node` with `key=A` and `value=detA` and inserts this `BST_Node` to the tree. **Hint:** The BST should still be sorted after insertion. Thus, new node should be inserted to the left sub-tree of root node if `A < root.key`. If `A == root.key`, only `root.value` field should be updated. Otherwise the new node should be inserted to the right sub-tree.

Finally, implement the recursive determinant function given in introduction.

Part 2 – Memoization with Hash Table [40 points]

In this part, you will implement a hash table and use it as the storage element for the determinant calculation function. A hash table item is defined as follows:

```
class HT_Item
{
    public :
        Matrix key;
        long value;
};
```

A hash table keeps an array of HT_Item pointers and uses a hash function to calculate an item's address inside the array. Array is initialized with NULLs, indicating that locations are empty. For this assignment, array size is 65536 elements. Implement the following hash function in the Matrix class:

```
function hash of Matrix M
    hash_value = M.N

    loop over all rows,  $0 \leq i < N$ 
        loop over all columns,  $0 \leq j < N$ 
            hash_value = 61*hash_value + M.element(i,j)

    return hash_value modulo 65536
```

A hash function is usually many-to-one (surjective). That is, hash of two distinct matrices could be equal to each other, which is called a collision. In this assignment, you will use linear probing for collision handling. That is, if a location is full, the next non-NULL location in the array will be used.

Implement a hash table with the following functions:

- `bool key_exists(Matrix A)`, that returns `true` if the table has an HT_Item with `key == A` and `false` otherwise. **Hint:** Start at location A's hash and search until an HT_Item with `key == A` or `NULL` found. You can assume no item will be deleted, thus you do not need to keep deleted flags.
- `long search(Matrix A)`, that returns the `value` field of the item whose `key == A`.
- `void insert(Matrix A, long detA)`, that dynamically creates a HT_Item with `key=A` and `value=detA` and inserts this HT_Item to the table. **Hint:** You can use a `count` variable that is initialized to 0 and incremented at every successful insertion. `count == 65536` means that hash table is full.

Finally, implement the recursive determinant function given in introduction.

Part 3 – Comparison of Two Methods [20 points]

In this part, you are going to read multiple matrices from different files and calculate their determinants. You are going to analyze these methods and compare them in terms of time and memory usage.

In the “matrices.zip” file, there are five folders for different sizes ($N = 11, 12, 13, 14, 15$) of matrices. Each folder contains five files for five different matrices with same size.

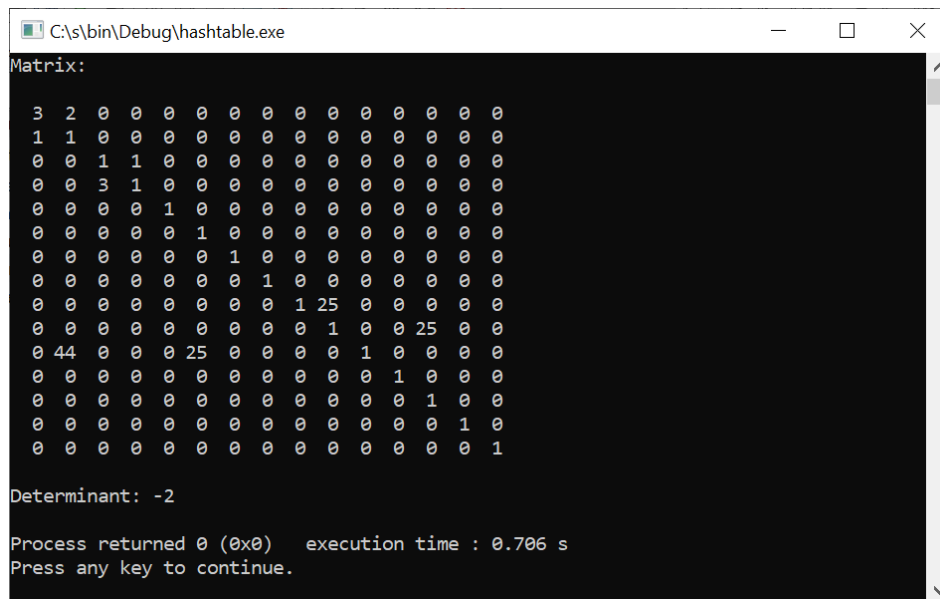
Using the given matrices, obtain:

- average time to compute the determinant
- average memory usage of the computation
 - for BST method, this is the number of `BST_Node` which are allocated times `sizeof(BST_Node)`
 - for Hash Table method, this is the number of `HT_Item` which are allocated times `sizeof(HT_Item)`

for changing values of N .

After obtaining these values, plot average time vs N and average memory usage vs N graphs for each method and comment on your findings. Put your graphs and commentary on your report.

Example run:



```
C:\s\bin\Debug\hashtable.exe
Matrix:
 3  2  0  0  0  0  0  0  0  0  0  0  0  0  0
 1  1  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  1  1  0  0  0  0  0  0  0  0  0  0  0
 0  0  3  1  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  1  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  1  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  1  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  1  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  1  25  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  1  0  25  0  0
 0 44  0  0  0 25  0  0  0  1  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  1  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  1  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  1  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  1

Determinant: -2

Process returned 0 (0x0)   execution time : 0.706 s
Press any key to continue.
```