IE523: Financial Computing
Fall, 2012
Programming Assignment 8: Repeated Squaring Algorithm
Due Date: 3 November, 2016
©Prof. R.S. Sreenivas

Later on in the course, when we deal with the problem of computing the price of an *European Option* efficiently using the method of *Dynamic Programming*, we will have to deal with the problem of taking an $n \times n$ matrix $\mathbf{A}$ and raising it to a large number. That is, we will have to compute $\mathbf{A}^k$ for large values of k.

You were introduced to this algorithm (although not by name) in Problem 2 of Quiz 2.

You could compute it by multiplying $\mathbf{A}$ over with it self k times. That is, (with *NEWMAT*) something along the lines of

```
1: C = A;
2: for int $i = 1$; $i < k$; $i$++ do
3:     C = A*C;
4: end for
```

Assuming you are doing straightforward matrix multiplication (nothing clever like Strassen's method, etc.) then the above procedure will take $O(n^3k)$ steps, as each matrix multiplication is $O(n^3)$ and there are k-many of them.

A candidate algorithm that is more efficient than the one shown above goes by the name of *Repeated Squaring*, and it described below.

Repeated Squaring Algorithm

Suppose you want to compute $\mathbf{A}^{11}$, you write the exponent 11 in binary – which is $\langle 1\ 0\ 1\ 1\rangle_2$. That is, $11 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$. You then compute all $\lceil \log_2 11 \rceil$-many powers of $\mathbf{A}$. That is, you compute $\mathbf{A}, \mathbf{A}^2, \mathbf{A}^4, \mathbf{A}^8$, then add the appropriate components as per the binary expansion of the exponent. In our case, we add $\mathbf{A}^8 \times \mathbf{A}^2 \times \mathbf{A}$ to get the final product. It turns out that these constituent powers of $\mathbf{A}$ (i.e. 8, 2 and 1) can be computed in a recursive manner quite efficiently. Here is something from the following link that computes n^k for integers n and k.

```
1: int power( int $n$, int $k$ )
2: if $k == 0$ then
3:     return 1
4: end if
5: if $k$ is odd then
6:     return ($n$ * power ($n \times n$, $\frac{(k-1)}{2}$))
7: else
8:     return (power( $n \times n$, $\frac{k}{2}$ ))
9: end if
```

A similar approach can be used to compute $\mathbf{A}^k$ for a square matrix $\mathbf{A}$. I leave the nitty-gritty details for you to figure out. Keep in mind that with NEWMAT on your side, the matrix operations are structurally the same as that for scalars.

Complexity of Repeated Squaring vs. Brute Force Multiplication

If you have to compute $\mathbf{A}^k$, and assuming each multiplication is $O(b^3)$ (nothing clever here, straightforward matrix multiplication), and since we have $\log_2 k$-many of these to do (or, since $\log_2 k = \frac{\log_{10} k}{\log_{10} 2}$ we can replace $\log_2 k$ with $O(\log k)$), this entire operation takes $O(b^3 \log k)$. This is in contrast to the $O(b^3 k)$ procedure for multiplying $\mathbf{A}$ with itself k times. Resulting in a total complexity of $O(b^3 \ln k)$. If we used *Strassen*'s method for matrix multiplication we would have a procedure that is $O(b^{2.81} \ln k)$.

The Programming Assignment

1. (Using *NEWMAT*) I want you to write a recursion routine

   ```
   Matrix repeated_squaring(Matrix A, int exponent, int no_rows)
   ```

   which takes a (no_rows $\times$ no_rows) matrix A and computes $\mathsf{A}^{\mathsf{exponent}}$ using the *Repeated Squaring Algorithm*.

2. Your code should be able to take as input the size and exponent as input on the command line. That is, if we want to compute $\mathbf{A}^k$, where $\mathbf{A}$ is an $(n \times n)$ square matrix, I want to be able to read n and k on the command-line. It should fill the entries of the matrix $\mathbf{A}$ with random entries in the interval $(-5, 5)$[1].

3. The output should indicate: (1) The number of rows/columns in $\mathbf{A}$ (that is read from the command line), (2) The exponent k (that is read from the command line), (3) The result and the amount of time it took to compute $\mathbf{A}^k$ using repeated squaring, and (4) The result and the amount of time it took to compute $\mathbf{A}^k$ using brute force multiplication. A sample output is shown in figure 1.

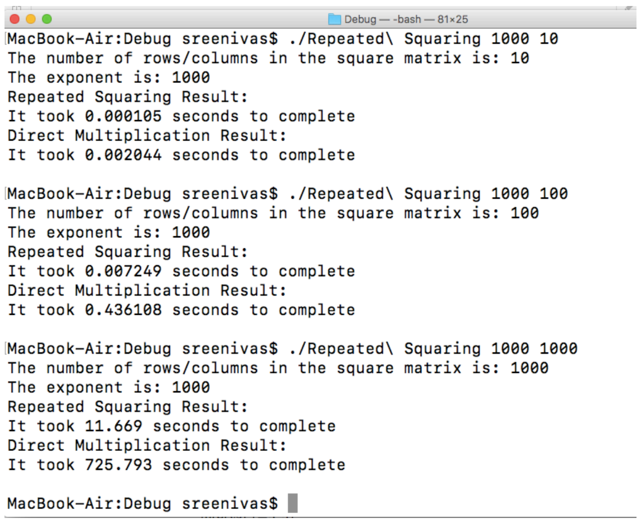
4. I want you to provide a plot of the computation time (in seconds) for the two methods as a function of the size of the matrix. That is, I am looking for something along the lines of figure 2. For this, you will have to place timer objects before and after appropriate portions of your code and do the needful – as the following lines of code illustrate.

   ```
   time_before = clock();
   B = repeated_squaring(A, exponent, dimension);
   time_after = clock();
   ```

[1]See strassen.cpp for ideas.

diff = ((float) time_after - (float) time_before);
cout << "It took " << diff/CLOCKS_PER_SEC << " seconds to complete" << endl;

In terms of the value of the exponent, tell me the regions where one algorithm performs better than the other, as far as computation time is concerned.



```
MacBook-Air:Debug sreenivas$ ./Repeated\ Squaring 1000 10
The number of rows/columns in the square matrix is: 10
The exponent is: 1000
Repeated Squaring Result:
It took 0.000105 seconds to complete
Direct Multiplication Result:
It took 0.002044 seconds to complete

MacBook-Air:Debug sreenivas$ ./Repeated\ Squaring 1000 100
The number of rows/columns in the square matrix is: 100
The exponent is: 1000
Repeated Squaring Result:
It took 0.007249 seconds to complete
Direct Multiplication Result:
It took 0.436108 seconds to complete

MacBook-Air:Debug sreenivas$ ./Repeated\ Squaring 1000 1000
The number of rows/columns in the square matrix is: 1000
The exponent is: 1000
Repeated Squaring Result:
It took 11.669 seconds to complete
Direct Multiplication Result:
It took 725.793 seconds to complete

MacBook-Air:Debug sreenivas$
```

Figure 1: A sample output for different exponents and matrix-dimensions.

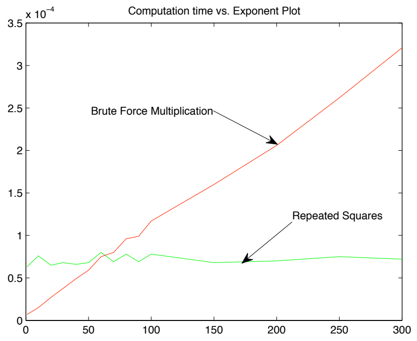


Figure 2: A comparison of the computation-time (obtained experimentally) for brute-force exponentiation and the method of repeated squares of a random 5×5 matrix as a function of the exponent.