Exponentiation and Java

Stephen Merrin

Abstract

We discuss an interesting teaching example for the Java programming language, where the problem is how to handle the operation of exponentiation. Specifically, suppose $b$ is some quantity that we want to raise to an integer power $n$. We address two issues. First, we ask if there is a more efficient exponentiation algorithm other than just iterating multiplication. Second, we wonder if can write the code just once to compute $b^n$ and have it work regardless of whether $b$ is an integer type, a floating point type, a square matrix, etc. The answers turn out to be yes in both cases. We will describe a strikingly efficient exponentiation algorithm well known to computer scientists but not so well known to mathematicians. We will use “abstract classes” in Java to write the essential exponentiation algorithm just once and have it work for all data types for which exponentiation makes sense.

1 Efficient exponentiation

Suppose you want to raise the quantity $b$ to the power $n$, where $n$ is a non-negative integer and $b$ is a number or a square matrix. The straightforward algorithm to compute $b^n$ requires $n - 1$ multiplications: $b^2 = b \cdot b$, $b^3 = b \cdot b \cdot b$, $b^4 = b \cdot b \cdot b \cdot b$, etc. Can we improve upon this method? Yes, we can. For example, to compute $b^4$ we could start with $b$ and then square twice: $b^4 = (b^2)^2$. On a calculator this is done simply by pressing the “squaring key” twice, which is a total of two multiplications. Similarly, we can compute $b^8$ as $b^8 = (b^4)^2$, which uses only three multiplications, a big improvement over the seven multiplications needed by the straightforward algorithm.

Our investigation so far suggests that to compute $b^n$ efficiently, we should do $b^n = (b^{n/2})^2$ when $n$ is even. What about the case when $n$ is odd? If $n$ is
odd, then \( n - 1 \) is even, and we can simply write \( b^n = b^{n-1} \cdot b = b^{\frac{n-1}{2}}^2 b \).

We have just stumbled upon an amazingly efficient recursive exponentiation algorithm:

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
\left(b^{\frac{n}{2}}\right)^2 & \text{if } n \text{ is even} \\
\left(b^{\frac{n-1}{2}}\right)^2 b & \text{if } n \text{ is odd} 
\end{cases}
\]

Let’s look at one more example. Our new method would compute \( b^{13} \) as \( \left((b^2 b)^2\right)^2 b \), a total of just five multiplications.

Just how efficient is this method? To analyze the complexity of this recursive algorithm, let’s define the function \( c(n) \) to be the number of multiplications required by our algorithm to compute \( b^n \). To compute \( b^n \) when \( n \) is even, we need one additional multiplication after computing \( b^{\frac{n}{2}} \), and so \( c(n) = c\left(\frac{n}{2}\right) + 1 \) when \( n \) is even. When \( n \) is odd, we need two more multiplications after computing \( b^{\frac{n-1}{2}} \), and so in this case \( c(n) = c\left(\frac{n-1}{2}\right) + 2 \). Altogether we have

\[
c(n) = \begin{cases} 
0 & \text{if } n = 0 \\
c\left(\frac{n}{2}\right) + 1 & \text{if } n \text{ is even} \\
c\left(\frac{n-1}{2}\right) + 2 & \text{if } n \text{ is odd} 
\end{cases}
\]

In the following table we have computed \( c(n) \) for a few values of \( n \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>( c(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>8</td>
</tr>
<tr>
<td>200</td>
<td>9</td>
</tr>
<tr>
<td>300</td>
<td>11</td>
</tr>
<tr>
<td>400</td>
<td>10</td>
</tr>
<tr>
<td>500</td>
<td>13</td>
</tr>
<tr>
<td>600</td>
<td>12</td>
</tr>
<tr>
<td>700</td>
<td>14</td>
</tr>
<tr>
<td>800</td>
<td>11</td>
</tr>
<tr>
<td>900</td>
<td>12</td>
</tr>
<tr>
<td>1000</td>
<td>14</td>
</tr>
</tbody>
</table>

The function \( c(n) \) grows very slowly. In fact, one can easily show by induction on \( n \) that \( c(n) \leq 2 \log_2 n \) for all \( n \). Contrast this with the \( n - 1 \) multiplications required by the naive exponentiation algorithm that simply iterates multiplication.
2 Java programs

We now have at our disposal an amazingly efficient algorithm to carry out exponentiation. Having dealt with the mathematical facets of the algorithm, interesting enough in their own right, we turn next to computer programming language issues. Our problem will be how we should go about implementing our algorithm in the Java programming language. An interesting question is the following: Can we somehow capture the essence of the algorithm by writing the core code just once and have it work for all meaningful data types? After all, we will be interested in computing $b^n$ for many different kinds of $b$'s: $b$ could be an integer type, a floating point type, a complex number, or even a square matrix. In short, $b$ could be an object of any type for which multiplication is defined.

We will describe how to set up a Java class hierarchy, with an “abstract class” at the root that captures our exponentiation algorithm, and child classes that implement the abstract methods for specific data types. By way of example, we provide child classes to handle exponentiation for type int, type BigInteger, and matrices. All the Java programs shown below were written and tested using version 1.1.6 of the JDK.

Below is the abstract class Exponentiate that contains our core exponentiation algorithm. This is the only time that we need to write this algorithm.

```java
public abstract class Exponentiate
{
    public abstract Object multiply(Object a, Object b);

    //return the multiplicative identity
    public abstract Object identity(Object a);

    //raise b to the n, n >= 0
    public Object power (Object b, int n)
    {
        if(n == 0) return identity(b); // b to the 0 is 1
        Object c = power(b, n/2);
        c = multiply(c, c);
        if(n % 2 == 0) return c; // case when n is even
        return multiply(c, b); // case when n is odd
    }
}
```
The abstract methods that must be implemented in all child classes of the class Exponentiate are:

- the “multiply” method that says how to multiply two objects
- the “identity” method that says what $b^0$ should be. If desired, we could omit this method altogether if we replace the base case $b^0 = 1$ of our exponentiation algorithm with $b^1 = b$. Nevertheless, for convenience and completeness we have included this optional method. For example, if $b$ is a square matrix, $b^0$ is the identity matrix of the same size as $b$.

In our first application, we show how to handle $b^n$ when $b$ is of type int. The IntPower class supplies the required multiply and identity methods, and IntPowerTest is a simple test program.

class IntPower extends Exponentiate {
    public Object multiply(Object c, Object d) {
        int a = ((Integer)c).intValue();
        int b = ((Integer)d).intValue();
        return new Integer(a * b);
    }
    public Object identity(Object a) {return new Integer(1);}
}

class IntPowerTest {
    public static void main(String[] args) {
        IntPower i = new IntPower();
        Integer b = new Integer(3);
        System.out.println("3 raised to the 10 is " + i.power(b, 10));
    }
}

The output of IntPowerTest is
3 raised to the 10 is 59049

4
Our second example demonstrates how to compute $b^n$ when $b$ is a BigInteger. BigIntegers are integer types with unlimited precision. Our test program computes $3^{1000}$.

```java
import java.math.BigInteger;
class BigIntPower extends Exponentiate {
    public Object multiply(Object c, Object d) {
        return ((BigInteger)c).multiply((BigInteger)d);
    }

    public Object identity(Object a) {
        return new BigInteger("1");
    }
}

class BigIntPowerTest {
    public static void main(String[] args) {
        BigIntPower i = new BigIntPower();
        BigInteger b = new BigInteger("3");
        System.out.println("3 raised to the 1000 is \n" + i.power(b,1000));
    }
}
```

If we run `BigIntPowerTest`, we get the following output:

3 raised to the 1000 is
1322070819480806636890455259752144365965422032752148167664920368226828
5973467048995407783138506080619639097776968725823559509545821006189118
6534272525795367402762022519832080387801477422896484127439040011758861
804112894781562309443806156617305408667449050617812548034405547054397
0388958174653682549161362208302685637785822902284163983078878969185564
0408489893760937324217184635993869551676501894058810906042608967143886
4102814350385648747165832010614366132173102768902855220001
Finally, we show how we can do matrix exponentiation. Assuming for the moment that we already have written a Matrix class that implements matrices and matrix multiplication, let’s look at the following child class of Exponentiate that does matrix exponentiation. Our own implementation of matrices appears in the Appendix.

class MatrixPower extends Exponentiate
{
    public Object multiply(Object a, Object b)
    { return ((Matrix)a).multiply((Matrix)b); }

    public Object identity(Object a)
    { return ((Matrix)a).identity(); }
}

class MatrixPowerTest
{
    public static void main(String[] args)
    {
        MatrixPower p = new MatrixPower();
        int[][] a = {{1, 3}, {-6, 7}};
        Matrix b = new Matrix(a);
        System.out.println("The matrix b is");
        b.print();
        System.out.println("The matrix b raised to the power 10 is");
        ((Matrix)(p.power(b,10))).print();
        System.out.println();
        int[][] c = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
        b = new Matrix(c);
        System.out.println("The matrix b is");
        b.print();
        System.out.println("The matrix b raised to the power 6 is");
        ((Matrix)(p.power(b,6))).print();
        System.out.println();
        System.out.println("The matrix b raised to the power 0 is");
        ((Matrix)(p.power(b,0))).print();
    }
}
The output of MatrixPowerTest is

The matrix \( b \) is
\[
\begin{pmatrix}
1 & 3 \\
-6 & 7
\end{pmatrix}
\]
The matrix \( b \) raised to the power 10 is
\[
\begin{pmatrix}
8176303 & 1476984 \\
-2953968 & 11130271
\end{pmatrix}
\]

The matrix \( b \) is
\[
\begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{pmatrix}
\]
The matrix \( b \) raised to the power 6 is
\[
\begin{pmatrix}
1963440 & 2412504 & 2861568 \\
4446414 & 5463369 & 6480324 \\
6929388 & 8514234 & 10099080
\end{pmatrix}
\]
The matrix \( b \) raised to the power 0 is
\[
\begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{pmatrix}
\]
3 Appendix

To make our examples self-contained, we include here our implementation of matrices that will work with the previous example. Our Matrix class provides a multiply method, a print method, and a method that returns an identity matrix of the right size.

```java
public class Matrix
{
    private int rows, cols;
    private int[][] array;

    public Matrix(int r, int c)
    {
        rows = r;
        cols = c;
        array = new int[rows][cols];
        int i, j;
        for(i = 0; i < rows; i++)
            for(j = 0; j < cols; j++)
                array[i][j] = 0;
    }

    public Matrix(int[][] a)
    {
        rows = a.length;
        cols = a[0].length;
        array = a;
    }

    public Matrix multiply(Matrix m) // multiply "this" times m
    {
        if( cols != m.rows )
        {
            System.out.println( "multiplication undefined" );
            return null;
        }
        Matrix p = new Matrix(rows, m.cols);
        // p is the product "this" times m
        int i, j, k, sum;
```
for(i = 0; i < rows; i++) // row i in this
    for(j = 0; j < m.cols; j++) // col j in m
    {
        sum = 0;
        for(k = 0; k < cols; k++)
            // (row i of this) times (col j of m)
                sum = sum + array[i][k] * m.array[k][j];
        p.array[i][j] = sum;
    }
    return p;
} // multiply

// return same size matrix as "this", with 1's down diagonal
public Matrix identity()
{
    Matrix i = new Matrix(rows, cols);
    int min = rows;
    if(rows > cols) min = cols;
    for(int j = 0; j < min; j++) i.array[j][j] = 1;
    return i;
}

public void print()
{
    int i, j;
    for(i = 0; i < rows; i++)
    {
        for(j = 0; j < cols; j++)
            System.out.print( array[i][j] + " " );
        System.out.println();
    }
} // print
} // Matrix