

CS303 (Spring 2008)— Solutions to Assignment 7

Problem 1

Here is our implementation of a class for long numbers. It includes addition, subtraction, and both ways of multiplying. The rest of the code, not included here, simply times the execution times for this implementation.

```
import java.util.Random;

public class Number
{
    int size;
    int [] a;
    int base;

    public Number (int n, int base, int fill)
    {
        Random r = new Random ();
        this.base = base;
        a = new int [n];
        for (int i = 0; i < n; i ++)
            if (fill == -1) a[i] = r.nextInt (base);
            else a[i] = fill;
        if (fill == -1) a[n-1] = 1+r.nextInt (base-1);
        if (fill == 0) size = 1; else size = n;
    }

    public Number (int [] b, int base)
    {
        this.base = base;
        size = b.length;
        while (size > 0 && b[size-1] == 0) size --;
        a = new int [size];
        for (int i = 0; i < size; i ++) a[i] = b[i];
    }

    public int digit (int i)
    { if (i >= size) return 0; else return a[i]; }

    public int size ()
    { return size; }

    public int base ()
    { return base; }

    public Number higher (int k)
    {
        int s = size - k;
        if (s < 0) s = 0;
    }
}
```

```

    int [] b = new int [s];
    for (int i = 0; i < s; i ++)
        b[i] = digit (k+i);
    return new Number (b, base());
}

public Number lower (int k)
{
    int [] b = new int [k];
    for (int i = 0; i < k; i ++)
        b[i] = digit (i);
    return new Number (b, base());
}

public Number shift (int s)
{
    int [] b = new int [size+s];
    for (int i = 0; i < size; i ++) b[i+s] = digit(i);
    return new Number (b, base);
}

public Number scale (int s)
{
    int [] b = new int [size+1];
    for (int i = 0; i < size; i ++) b[i] = 0;
    for (int i = 0; i < size; i ++)
        { b[i] += s*digit(i); b[i+1] += b[i]/base; b[i] %= base; }
    return new Number (b, base);
}

public Number add (Number y)
{
    int n;
    if (size() > y.size()) n = size(); else n = y.size();
    int [] b = new int [n+1];
    for (int i = 0; i < n+1; i ++) b[i] = 0;
    for (int i = 0; i < n; i ++)
        { b[i] += digit(i) + y.digit(i); b[i+1] += b[i]/base; b[i] %= base; }
    return new Number (b, base);
}

public Number subtract (Number y)
{
    int n;
    if (size > y.size()) n = size; else n = y.size();
    int [] b = new int [n+1];
    for (int i = 0; i < n+1; i ++) b[i] = 0;
    for (int i = 0; i < n+1; i ++)
        { b[i] += digit(i) - y.digit(i); if (b[i] < 0) { b[i+1] --; b[i] += base; }}
    return new Number (b, base);
}

```

```

public Number regularMult (Number y)
{
    Number z = new Number (1, base(), 0);
    for (int i = 0; i < size(); i ++)
        z = z.add (y.shift(i).scale (digit(i)));
    return z;
}

public Number DCMult (Number y)
{
    if (size() <= 1) return y.scale (digit(0));
    else if (y.size() <= 1) return scale (y.digit(0));
    else
        {
            int n;
            if (size () > y.size()) n = size(); else n = y.size();
            Number x1 = higher (n/2), x2 = lower (n/2),
                y1 = y.higher (n/2), y2 = y.lower (n/2);
            Number ah = x1.DCMult (y1), al = x2.DCMult (y2),
                am = (x1.add(x2)).DCMult(y1.add(y2));
            am = am.subtract(al).subtract(ah);
            return ah.shift(2*(n/2)).add(am.shift(n/2)).add(al);
        }
}

public void print ()
{
    if (size == 0) System.out.print ("0");
    else for (int i = size - 1; i >= 0; i --) System.out.print (digit(i));
}
}

```

The figure below shows the running time plots for both. Notice that the time is on a log scale, so the difference is actually quite significant. For arrays of 10000 digits, the Divide&Conquer algorithm was about 5 times faster already. In our implementation, the break-even point between the two algorithms, when running base 2, was at about 250 digits.

Exercise 28.2-4

The key insight here is the following view of Strassen's Algorithm: by computing a product of two 2×2 matrices with only 7 multiplications, it can apply this to make 7 recursive calls while halving the matrix dimensions. This generalizes: if we can multiply $m \times m$ matrices with just k basic multiplications, we can also use it to solve $n \times n$ instances with k recursive calls to n/m sized matrices.

Plugging in the values from the exercise, this will thus gives us Divide&Conquer algorithms with the following three running times:

$$\begin{aligned}
 T(n) &= 132464 \cdot T(n/68) \\
 T(n) &= 143640 \cdot T(n/70) \\
 T(n) &= 155424 \cdot T(n/72).
 \end{aligned}$$

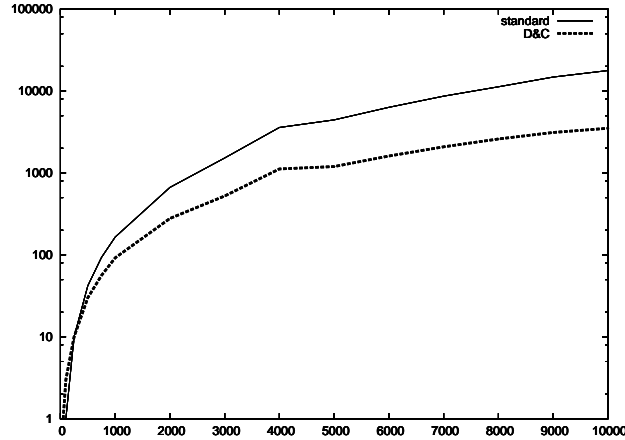


Figure 1: Running times of the standard and the Divide&Conquer algorithms

Solving these using the Master Theorem (all three fall into case 1) gives us

$$\begin{aligned}
 T(n) &= \Theta(n^{\log_6 8132464}) = \Theta(n^{2.7951285\dots}) \\
 T(n) &= \Theta(n^{\log_7 0143640}) = \Theta(n^{2.7951227\dots}) \\
 T(n) &= \Theta(n^{\log_7 2155424}) = \Theta(n^{2.7951474\dots})
 \end{aligned}$$

On the other hand, the running time of Strassen's algorithm is $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.8073549})$. So at least asymptotically, all of these algorithms outperform Strassen's. Among the three, the one using 70×70 matrices is asymptotically fastest. Of course, given how slight the differences are, in practice, many other issues will matter.

Exercise 28.2-6

The solution to this problem is actually quite similar to what we did for integer multiplication. If the inputs are $a + bi$ and $c + di$, the output is $ac - bd$ and $ad + bc$. Using two real multiplications, we can determine ac and bd . Then, we can observe that $ad + bc = (a + b)(c + d) - ac - bd$, so the third real multiplication we do is to compute $(a + b)(c + d)$, from which we then subtract the other two products to obtain $ad + bd$.

Problem 4

First off, we notice that we could easily solve this problem in $O(n^2)$ calls to `areEqual`, simply by comparing each pair of array entries. But of course, we want a faster solution.

The crucial observation here for a divide-and-conquer approach is that if we divide the array into two parts of sizes $n/2$ each, and some entry is equal to at least $n/2$ other entries, then it must be equal to at least $n/4$ of the entries in one of the halves. Hence, we can use a recursive invocation to get (at most two) candidates, and then test in linear time whether they appear $n/2$ times overall. The resulting recursive function is given below.

Initially, we invoke `Majority(1, n)`. The correctness of the algorithm is captured by the following lemma:

Lemma 1 *The set S returned by `Majority(l, r)` is exactly the set of all entries that appear at least $1 + \lceil \frac{r-l}{2} \rceil$ times.*

Proof. We prove this by induction on $r - l$. If $r = l$, then the correctness is obvious, as we return just the one entry, which appears once.

Algorithm 1 Majority (l, r)

```
1: if  $l = r$  then
2:   return  $\{l\}$ .
3: else
4:   Let  $m = \lfloor \frac{l+r}{2} \rfloor$ .
5:   Let  $S = \text{Majority}(l, m) \cup \text{Majority}(m+1, r)$ .
6:   for each  $i \in S$  do
7:     Compute areEqual(a, i, j) for  $j = l \dots r$ .
8:     If  $a[i]$  is equal to at most  $\lceil \frac{r-l}{2} \rceil$  of those  $r-l+1$  entries  $a[j]$ , remove it from  $S$ .
9:   end for
10:  return  $S$ .
11: end if
```

For larger values of $r-l$, first notice that we never return any entries that do not appear as often as we claim in the lemma. That's because we explicitly remove them from S in step 8 of the algorithm. Notice that this also proves that the set S returned contains either 0 or 1 entries in each case, because no two distinct entries can be equivalent to more than half of the entries each.

So we only need to prove that indeed, we do return an entry if it is equivalent to at least $1 + \lceil \frac{r-l}{2} \rceil$ entries (including itself). Suppose that i is such an entry. Then $a[i]$ must be equivalent to at least $1 + \lceil \frac{m-l}{2} \rceil$ among the entries $a[l], \dots, a[m]$, or to at least $1 + \lceil \frac{r-(m+1)}{2} \rceil$ entries among $a[m+1], \dots, a[r]$. For if it were not, then it could be equal to at most a total of $\lceil \frac{m-l}{2} \rceil + \lceil \frac{r-(m+1)}{2} \rceil < \lceil \frac{r-l}{2} \rceil$ entries between $l \dots r$, a contradiction. Hence, by Induction Hypothesis, i must be in at least one of $\text{Majority}(l, m)$ and $\text{Majority}(m+1, r)$, and hence is found in the subsequent check. ■

For the running time, we notice that the running time satisfies $T(k) = 2T(k/2) + \Theta(k)$ (because the final scan only takes linear time, and all the rest of the local computation is constant time), which we know to have solution $T(k) = \Theta(k \log k)$ (e.g., MergeSort). In particular, the running time is $\Theta(n \log n)$ for the full problem.