# Assignment 11 Solution

## Program Analysis for Power Computing

This problem asks you to establish the correctness of, and measure the empirical running time of two different methods for finding powers under two different data assumptions, and compare these real times with the big-O running times that you obtain analytically.

The algorithms are the straightforward ("regular") power-computing algorithm:

```
static double regDoublePower(long base, long N)
  {
  double result = 1.0;

  while (N > 0)
    {
    result = result * base;

    --N;
    }

  return result;
  }
```

vs. a power-computing algorithm called the "Russian Peasants" (rp) method:

```
static double rpDoublePower(long base, long N)
  {
  double s = base;

  double result = 1.0;

  while (N > 0)
    {
    result = (N%2 == 1 ? result * s : result);

    N = N/2;

    s = s * s;
    }

  return result;
  }
```

**Problem 1:** [10/50 points]

Prove that regDoublePower is partially correct with respect to:

Input assertion: $N == N_0 \wedge N_0 \geq 0$

Output assertion: $result == b^{N0}$.

**Solution 1:**

As the **loop invariant**, use: $result*b^{N} == b^{N0} \wedge N \geq 0$

We will also use the observation that b never changes and treat b as a constant.

Use **transition induction** to establish the invariant. Toward this end, there are three verification conditions to be established:

- **(Input assertion $\wedge$ Initialization) $\Rightarrow$ Invariant**

  $(N == N_0 \wedge N_0 \geq 0 \wedge result == 1) \Rightarrow (result*b^{N} == b^{N0} \wedge N \geq 0)$

  **Proof**: Assuming the left-hand-side of $\Rightarrow$, the right-hand side is equivalent to $(1*b^{N0} == b^{N0} \wedge true)$, which simplifies to true.

- **(Invariant $\wedge \neg$Test) $\Rightarrow$ Output assertion**

  $((result*b^{N} == b^{N0} \wedge N \geq 0) \wedge \neg (N > 0)) \Rightarrow result == b^{N0}$

  **Proof**: Assuming the left-hand side of $\Rightarrow$, $(N \geq 0 \wedge \neg (N > 0))$ implies $N == 0$. then $result*b^{0} == b^{N0}$ gives $result == b^{N0}$.

- **(Invariant $\wedge$ Test $\wedge$ Body) $\Rightarrow$ Invariant'**

  $((result*b^{N} == b^{N0} \wedge N \geq 0) \wedge (N > 0) \wedge (result' == result*b) \wedge (N' == N - 1)) \Rightarrow$

  $(result' *b^{N'} == b^{N0} \wedge N' \geq 0)$

  **Proof:** Assuming the left-hand side of $\Rightarrow$, the right-hand side is equivalent to $result* b *b^{N-1} == b^{N0}$ which simplifies to $result* b^{N} == b^{N0}$, which is implied by the left-hand side. The second conjunct on the right-hand side follows from $N' + 1 > 0$ on the left-hand side because the domain is that of integers.

**Problem 2:** [15/50 points]

Prove that rpDoublePower is partially correct with respect to:

Input assertion: $N == N_0 \land N_0 \geq 0$

Output assertion: result == $b^{N0}$.

**Solution 2:**

As the **loop invariant**, use: result * $s^N$ == $b^{N0}$ $\land$ $N \geq 0$

We will also use the observation that b never changes and treat b as a constant.

Use **transition induction** to establish the invariant. Toward this end, there are three verification conditions to be established:

- **(Input assertion $\land$ Initialization) $\Rightarrow$ Invariant**

  ($N == N_0 \land N_0 \geq 0 \land$ result == 1 $\land$ s == b) $\Rightarrow$ (result * $s^N$ == $b^{N0}$ $\land$ $N \geq 0$)

  **Proof**: Assuming the left-hand-side of $\Rightarrow$, the right-hand side is equivalent to (1 * $b^{N0}$ == $b^{N0}$ $\land$ $N_0 \geq 0$), which simplifies to true.

- **(Invariant $\land$ $\neg$Test) $\Rightarrow$ Output assertion**

  ((result * $s^N$ == $b^{N0}$ $\land$ $N \geq 0$) $\land$ $\neg$ (N > 0)) $\Rightarrow$ result == $b^{N0}$

  **Proof**: Assuming the left-hand side of $\Rightarrow$, we have ($N \geq 0$ $\land$ $\neg$ (N > 0)), which implies N == 0, so result * $s^N$ == result * $s^0$ == result * 1 == result, thus result == $b^{N0}$.

- **(Invariant $\land$ Test $\land$ Body) $\Rightarrow$ Invariant'**

  ((result * $s^N$ == $b^{N0}$ $\land$ $N \geq 0$) $\land$ $\neg$ (N > 0) $\land$ (result' == N %2 == 1? result*s : result) $\land$ (N' ==N/2) $\land$ (s' == $s^2$)) $\Rightarrow$ (result' * $s'^{N'}$ == $b^{N0}$ $\land$ $N' \geq 0$)

  **Proof:** Assume the left-hand side of $\Rightarrow$. Since N > 0, we have $N/2 \geq 0$, which is the second conjunct. To prove the first conjunct, we see that it is equivalent to

  (N %2 == 1? result*s : result) * $(s^2)^{N/2}$== $b^{N0}$, which is equivalent to

  (N %2 == 1? result*s*$(s^2)^{N/2}$ : result*$(s^2)^{N/2}$) == $b^{N0}$.

  If N %2 == 1, then the equality to be shown is:

  result*s*$(s^2)^{N/2}$ == $b^{N0}$,

  while if N %2 == 0, the equality is

$$result*(s^2)^{N/2} == b^{N0}.$$

**If N %2 == 1**, then N/2 == (N-1)/2, where division on the right is non-truncating. Thus the equality reduces to:

$$result*s*(s^2)^{(N-1)/2} == b^{N0}$$

which simplifies to:

$$result*s^N == b^{N0}$$

which follows from the left-hand side.

**If N %2 == 0**, the equality reduces to:

$$result* (s^2)^{N/2} == b^{N0}$$

where division is non-truncating, so this simplifies to

$$result*s^N == b^{N0}$$

which follows from the left-hand side as before.

**Problem 3:** [15/50 points]

We will examine four methods for finding powers of some integer base (`b`). The running time of each will be expressed in terms of the size of the exponent, i.e., the power, N. For simplicity, we will always use the same base, 7 and we will always use values of the power, N that are $2^i$ for some `i`. That is, the powers will themselves be powers of 2.

Using asymptotic analysis, we are to find the big-O running time of the two basic algorithms (regular and Russian Peasants method) in terms of `N` (the power to which the base is raised), using the number 7 as the value of b in both cases.

**Assumptions:**

For `double`, assume that the multiplication time is constant. For `BigInteger`, assume that the multiplication time is proportional to the product of the lengths of numbers being multiplied (as it would be for elementary school multiplication).

**Solution**:

For the `double` representation, we assume that the multiplication time is constant regardless of the operands. In this case, we have:

**Regular double**: [2/15 points] Since there are N iterations, and the loop body is constant-time, a bound is **O(N).**

**Russian Peasants double**: [3/15 points] Since there are log N iterations, and the loop body is constant-time, **O(log N)**.

**Regular BigInteger**: [4/15 points] There are N iterations, and the loop body requires time O(N) itself, because we are multiplying b in each case by increasingly longer multipliers, namely the value of result, which increases linearly as the algorithm progresses. So the overall time taken is proportional to (1 + 2 + 3 + … + N), which we know to be $\mathbf{O(N^2)}$.

**Russian Peasants BigInteger**: [6/15 points] There are log N iterations. The run-time of the loop body is dominated by the multiplication s = s*s; The number of digits in s *doubles* each iteration. Thus the overall time it takes is bounded by some constant c times (1 + 2*2 + 4*4 + 8*8 + … ), which is ($4^0 + 4^1 + 4^2 + 4^3 + \ldots$ ), where the last power is $4^{\log N} == N^2$. Thus the sum is $\mathbf{O(N^2)}$, since the last term dominates the *sum* of the preceding terms. To see this dominance, represent the sum as a base-4 integer:

1 . . . 1 1 1 1

$4^{\log N} + \ldots + 4^6 + 4^2 + 4^1 + 4^0$

The sum of the numbers represented by the low-order 1's is no greater than the number represented by the single high-order 1, so the overall sum is bounded by $2*4^{logN}$ □ **$O(N^2)$**.

Note that we could cut the execution time down substantially by not computing the final s = s*s;, since once N becomes 0, the value of s is no longer needed. A loop break statement could be used for this purpose, for example.

**Discussion**

The surprising result for BigInteger is the Russian Peasants doesn't improve the asymptotic performance over the regular method as it might be expected to do: both RP and the regular method are **$O(N^2)$**. However, it may still be better in terms of the multiplicative constant.

**Problem 4:** [10/50 points]

Describe how well the empirical evidence supports the big-O running times you found in 3.

**Solution**:

**Note**: I re-ran the testing program with 10 times the number of repetitions, since Java has gotten faster since the last time I ran it.

For **Regular Double** [2/15 points], the tightest upper bound appears to be **O(N)**. All values of T/N are around 2.2E-04 or less. Note that log N is clearly not also an upper bound, and N log N seems to be a loose upper bound.

| power: N | T/1 | T/logN | T/N | T/(N log N) | T/(N*N) |
|---|---|---|---|---|---|
| RegularDouble | | | | | |
| 16 | +6.400E-03 | +2.308E-03 | +4.000E-04 | +1.000E-04 | +2.500E-05 |
| 32 | +7.000E-03 | +2.020E-03 | +2.188E-04 | +4.375E-05 | +6.836E-06 |
| 64 | +1.360E-02 | +3.270E-03 | +2.125E-04 | +3.542E-05 | +3.320E-06 |
| 128 | +2.820E-02 | +5.812E-03 | +2.203E-04 | +3.147E-05 | +1.721E-06 |
| 256 | +5.560E-02 | +1.003E-02 | +2.172E-04 | +2.715E-05 | +8.484E-07 |

For **Russian Peasants Double** [2/15 points], **O(log N)** is the tightest upper bound. All values of T/log N are less that 1.5E-03.

| power: N | T/1 | T/logN | T/N | T/(N log N) | T/(N*N) |
|---|---|---|---|---|---|
| RpDouble | | | | | |
| 16 | +4.130E-03 | +1.490E-03 | +2.581E-04 | +6.453E-05 | +1.613E-05 |
| 32 | +4.710E-03 | +1.359E-03 | +1.472E-04 | +2.944E-05 | +4.600E-06 |
| 64 | +5.570E-03 | +1.339E-03 | +8.703E-05 | +1.451E-05 | +1.360E-06 |
| 128 | +6.240E-03 | +1.286E-03 | +4.875E-05 | +6.964E-06 | +3.809E-07 |
| 256 | +6.810E-03 | +1.228E-03 | +2.660E-05 | +3.325E-06 | +1.039E-07 |

**Both are of the above are consistent with our white-box analysis.**

For **Regular BigInteger** [3/15 points], the only upper bound appears to be **O($N^2$)**, settling toward a ratio of about 2.5 E-05. All other ratios are increasing.

| power: N | T/1 | T/logN | T/N | T/(N log N) | T/(N*N) |
|---|---|---|---|---|---|
| RegularBigInt | | | | | |
| 16 | +4.000E-02 | +1.443E-02 | +2.500E-03 | +6.250E-04 | +1.562E-04 |
| 32 | +8.000E-02 | +2.308E-02 | +2.500E-03 | +5.000E-04 | +7.812E-05 |
| 64 | +2.100E-01 | +5.049E-02 | +3.281E-03 | +5.469E-04 | +5.127E-05 |
| 128 | +6.200E-01 | +1.278E-01 | +4.844E-03 | +6.920E-04 | +3.784E-05 |
| 256 | +2.050E+00 | +3.697E-01 | +8.008E-03 | +1.001E-03 | +3.128E-05 |
| 512 | +7.570E+00 | +1.213E+00 | +1.479E-02 | +1.643E-03 | +2.888E-05 |
| 1024 | +2.816E+01 | +4.063E+00 | +2.750E-02 | +2.750E-03 | +2.686E-05 |
| 2048 | +1.065E+02 | +1.397E+01 | +5.199E-02 | +4.727E-03 | +2.539E-05 |
| 4096 | +4.225E+02 | +5.080E+01 | +1.032E-01 | +8.596E-03 | +2.518E-05 |
| 8192 | +1.674E+03 | +1.858E+02 | +2.044E-01 | +1.572E-02 | +2.495E-05 |
| 16384 | +6.686E+03 | +6.890E+02 | +4.081E-01 | +2.915E-02 | +2.491E-05 |

For **Russian Peasants BigInteger** [3/15 points], the only upper bound appears to be **O(N$^2$)**, settling toward a ratio of 3.37E-06. All other ratios are gradually increasing.

| power: N | T/1 | T/logN | T/N | T/(N log N) | T/(N*N) |
|---|---|---|---|---|---|
| RpBigInt | | | | | |
| 16 | +2.000E-02 | +7.213E-03 | +1.250E-03 | +3.125E-04 | +7.812E-05 |
| 32 | +8.000E-02 | +2.308E-02 | +2.500E-03 | +5.000E-04 | +7.812E-05 |
| 64 | +4.000E-02 | +9.618E-03 | +6.250E-04 | +1.042E-04 | +9.766E-06 |
| 128 | +9.000E-02 | +1.855E-02 | +7.031E-04 | +1.004E-04 | +5.493E-06 |
| 256 | +2.900E-01 | +5.230E-02 | +1.133E-03 | +1.416E-04 | +4.425E-06 |
| 512 | +9.100E-01 | +1.459E-01 | +1.777E-03 | +1.975E-04 | +3.471E-06 |
| 1024 | +3.550E+00 | +5.122E-01 | +3.467E-03 | +3.467E-04 | +3.386E-06 |
| 2048 | +1.403E+01 | +1.840E+00 | +6.851E-03 | +6.228E-04 | +3.345E-06 |
| 4096 | +5.700E+01 | +6.853E+00 | +1.392E-02 | +1.160E-03 | +3.397E-06 |
| 8192 | +2.238E+02 | +2.484E+01 | +2.732E-02 | +2.102E-03 | +3.335E-06 |
| 16384 | +9.069E+02 | +9.346E+01 | +5.535E-02 | +3.954E-03 | +3.378E-06 |

**Both are of the above are also consistent with our white-box analysis.**

We notice that for BigInteger, Russian Peasants still tends to be faster, by a factor of almost 10, but they have the same asymptotic growth rate.

It is somewhat unfair to compare the double versions with the BigInteger versions, since the latter have to do a lot more work in preserving all digits of the answer.