Journal of Uncertain Systems
Vol.1, No.3, pp.178-184, 2007
Online at: www.jus.org.uk

Russian Peasant Multiplication Algorithm, RSA Cryptosystem, and a New Explanation of Half-Orders of Magnitude*

J. Ivan Vargas[1] and Olga Kosheleva[2]
[1]*Department of Computer Science*
[2]*Department of Teacher Education*
University of Texas at El Paso
500 W. University, El Paso, TX 79968, USA

Received 24 February 2007; Accepted 10 March 2007

Abstract

In his papers, J. Hobbs has observed that when people make crude estimates, they usually feel reasonably comfortable choosing between alternatives which differ by a half order of magnitude (HOM). He also provided an explanation for this level of granularity based on the need for the resulting crude estimates to represent both the original data and the result of processing this data. According to this explanation, HOM are optimal – when we limit ourselves to these first crude estimates. In many practical situations, we do not stop with the original estimate, we refine it one or more times by using granules of smaller and smaller size. In this paper, we show that the need to optimally process such refined estimates leads to the same HOM granularity. Thus, we provide a new explanation for this level of granularity. © 2007 World Academic Press, UK. All rights reserved.

Half-orders of magnitude: empirical fact. People often need to make crude estimates of a quantity, e.g., estimating the size of a crowd or someone's salary. In [4, 5, 6], it was observed that when people make these crude estimates, they usually feel reasonably comfortable choosing between alternatives which differ by a half order of magnitude (HOM).

For example, a person can reasonably estimate whether the size of a crowd was closer to 100, or to 300, or to 1000. If we ask for an estimate on a more refined scale, e.g., 300 or 350, people will generally be unable to directly come up with such estimates. On the other hand, if we ask for an estimate on a coarser scale, e.g., 100 or 1000, people may be able to answer, but they will feel their answer is uninformative.

An interesting example of HOM is presented by coinage and currency. Most countries have, in addition to denominations for the powers of ten, one or two coins or bills between every two powers of ten. Thus, in the United States, in addition to coins or bills for \$.01, \$.10, \$1.00, \$10.00, and \$100.00, there are also coins or bills in common use for \$.05, \$.25, \$5.00, \$20.00, and \$50.00. These latter provide rough HOM measures for monetary amounts.

Half-orders of magnitude: the existing explanation. In [5, 6], an explanation for this level of granularity based on the need for the resulting crude estimates to represent both the original data and the result of processing this data. According to this explanation, HOM are optimal – when we limit ourselves to these first crude estimates.

*Corresponding author: olgak@utep.edu

Towards a new explanation. In many practical situations, we do not stop with the original estimate, we refine it one or more times by using granules of smaller and smaller size. In this paper, we show that the need to optimally process such refined estimates leads to the same HOM granularity. Thus, we provide a new explanation for this level of granularity.

Estimating vs. data processing: main difference. Estimation is a one-time process which provides a crude estimate for the quantity of interest. In many practical situations, this estimate is quite sufficient for decision making.

In other situations, however, the original crude estimate is not sufficient, and we must refine it. Let us describe this refinement in precise terms.

Refined estimates: a description. What does it mean to have a value m as a granularity level? Crudely speaking, this means that we consider granules of the sizes 1, m, m^2, ..., m^k, ...

A rough estimate means that we simply compare the actual value v with the sizes of these granules. The largest granule m^k for which $m^k \le v$ is then used as a rough estimate of the quantity v: $m^k \le v < m^{k+1}$. This rough-estimate granule means that we can estimate v from below by using granules of size m^k, but not by using larger granules.

Once we know that the granules of size m^k *can* be used to estimate v, a natural next question is *how many* granules of this size we can fit within v. Of course, we can only have $c_k < m$ granules. (Otherwise, we would be able to fit $m \cdot m^k$ values in v, and we would have $v \ge m^{k+1}$, i.e., we would conclude that the next granule also fits within v – contrary to our choice of m^k as the largest granule that fits within v.) So, in this next approximation, we are looking for the value $c_k < m$ for which $c_k \cdot m^k \le v < (c_k + 1) \cdot m^k$. The resulting value $c_k \cdot m^k$ – i.e., the size k plus the value c_k – provides a more accurate description of v than simply the size k of the largest granule.

The difference between the actual value v and the estimate $c_k \cdot m^k$ cannot be fitted with granules of size m^k. Thus, to get an even more accurate description of v, we must use granules of next smaller size m^{k-1} to cover this difference. In other words, we must find the largest value c_{k-1} for which $c_{k-1} \cdot m^{k-1}$ is contained in the difference $v - c_k \cdot m^k$, i.e., for which $c_{k-1} \cdot m^{k-1} \le v - c_k \cdot m^k < (c_{k-1} + 1) \cdot m^{k-1}$. This is equivalent to selecting c_{k-1} for which

$$c_k \cdot m^k + c_{k-1} \cdot m^{k-1} \le v < c_k \cdot m^k + (c_{k-1} + 1) \cdot m^{k-1}.$$

A further refinement of this estimate means that we use granules of even smaller size m^{k-2} to estimate the difference between the actual value v and the estimate-so-far $c_k \cdot m^k + c_{k-1} \cdot m^{k-1}$, etc. One can see that this refined estimation process leads to an m-ary representation of integers:

$$v = c_k \cdot m^k + c_{k-1} \cdot m^{k-1} + \ldots + c_1 \cdot m^1 + c_0.$$

Example. For example, to represent the number $v = 256$ with decimal granules 1, $m = 10$, 100, 1000, etc., we first find the largest granule which fits within 256 – the granule 100. This granule is our first (order-of-magnitude) representation of the number 256.

To get a better representation, we can describe how many times this granule fits within 256, i.e., approximate 256 as $2 \cdot 100$.

To get an even more accurate representation, we need to use granules of next smaller size 10 to represent the difference $256 - 200 = 56$ between the original number 256 and its approximate value 200. We can fit this granule 5 times, so we get an approximation $5 \cdot 10$ for the difference and correspondingly, the approximation $2 \cdot 100 + 5 \cdot 10 = 250$ for the original number 256. With this approximation, we still have an un-approximated difference $256 - 250 = 6$.

To get a more accurate approximation, we use the granules of smaller size 1. Within 6, this granule fits 6 times, so we get a representation $2 \cdot 100 + 5 \cdot 10 + 6 \cdot 1$ for the original number.

Conclusion: selecting granularity level means, in effect, selecting a base for number representation. The above general description and example both show that the use of a certain granule size m means, in effect, that we use m-ary system to represent numbers.

Which value m is the best for m-ary number representation? In view of the above observation, the question of which granule size is the best can be reformulated as follows: for which m the m-ary representation is the best?

Aren't binary numbers the best? They are used in computers. Normally, people use decimal numbers, with $m = 10$, and computers use binary numbers, with $m = 2$. It may seem that the fact that well-designed and well-optimized computational devices such as computers use binary numbers is an indication that (at least empirically) $m = 2$ is the best choice.

However, this is not necessarily true. The computer engineering choice of $m = 2$ is largely motivated by specific electronic hardware technologies, in which it is easier to manufacture an electronic switch with 2 possible states than with 3 or 10. Our objective is to explain *human* behavior, and for human data processing, these hardware considerations do not apply.

Binary numbers have been used in human data processing as well: Russian peasant multiplication algorithm. Binary numbers for electronic computers are a recent (20 century) phenomenon. However, it is worth mentioning that binary numbers were, in effect, used in data processing for several millennia. According to [7], binary-related algorithm for multiplication was used by ancient Egyptian mathematicians as early as 1800 B.C.E. This method is called *Russian peasant multiplication algorithm* because it was first observed in the 19 century by the Western visitors to Russia – where this method was widely used by the common folks (i.e., mainly peasants) [1, 7]. Later, a similar method was found (and decoded) in an ancient Egyptian papyrus.

This algorithm is especially useful if we want to multiply different numbers x by a given number n. This happens, e.g., if a merchant wants to compute the prices of different amounts of the item that he is selling: in this example, n is the price of a single item, and x is the number of such items.

In this procedure, we first transform the fixed number n into the binary code, i.e., represent n as a sum of powers of two. Interestingly, the transition to binary code was performed in the ancient Egypt in exactly the same way as it is done now: by sequentially dividing a number by 2 and then reading the remainders from bottom up.

Once such a binary representation is found, we can compute the product $n \cdot x$ as follows:

- first, we add x to itself, resulting in $2x$;
- then, we add $2x$ to itself, resulting in $4x = 2^2 \cdot x$;
- after that, we add $2^2 \cdot x$ to itself, then getting $8x = 2^3 \cdot x$, etc.
- once we have the values $2^i \cdot x$, we add those values which correspond to the representation of n as the sum of powers of 2, thus getting $n \cdot x$.

Example. For example, $n = 13$ is represented in binary code as $1101_2 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1$. For $n = 13$, the conversion to binary is performed as follows:

```
13 / 2 = 6 rem 1
 6 / 2 = 3 rem 0
 3 / 2 = 1 rem 1
 1 / 2 = 0 rem 1
```

Reading remainders from bottom up, we get the binary representation 1101_2.

Now, to compute $13x$, we consequently compute $2x$, $4x$, $8x$, and then add $x + 4x + 8x$.

This method is often faster than using decimal numbers. To compute $13x$, we need 3 additions (namely, doubling) to compute all three powers of two, and then 2 more additions to compute $x + 4x$ and then $13x$ as $(x + 4x) + 8x$. Overall, we need 5 additions.

This number is much smaller than what we would have needed if we decided to reduce multiplication to addition in the standard decimal representation, in which we would need to compute x, $2x$, $3x$, ..., $10x$, and then add $3x + 10x$, to the overall of 11 additions.

A similar method is used in cryptosystems. The efficiency of binary-based multiplication prompted the use of a similar technique in cryptosystems. In particular, in the most widely used RSA techniques (see, e.g., [2]), techniques which are used every time we access a secure webpage or make financial transactions online. Cryptosystems make computer communications secure by encoding messages, largely by raising a number x (representing a message) to a given power n (to be more precise, they compute the power x^n modulo some large number N). The efficiency of RSA and similar cryptosystems is based on the fact that it is computationally efficient to compute x^n but (unless we know factors of N) it is very computationally difficult to recover x from the transmitted message $M \stackrel{\text{def}}{=} x^n$. This exponentiation is time-consuming, it forms the dominant part of cryptoalgorithms running time; see, e.g., [3]. So, to make cryptosystems more efficient, it is important to compute x^n fast.

At present, exponentiation is mainly done by using the binary representation of n. Namely, we use multiplication to compute $x^2 = x \cdot x$, $x^4 = x^2 \cdot x^2$, $x^8 = x^4 \cdot x^4$, ..., and then we multiply the powers corresponding to the powers of 2 that are present in the binary expansion of n.

For example, to compute x^{13}, we compute x^2, x^4, x^8, and then multiply $x \cdot x^4 \cdot x^8$. Overall, just like we need 5 additions to multiply a given number by 13, we need 5 multiplications to raise a given number x to the 13-th power.

Binary-based methods are widely used but they are not always optimal. In practice, binary techniques are so much faster than decimal-based ones that it was originally conjectured that they are optimal for all n. Specifically, it was conjectured that if we want to compute a product $n \cdot x$ by using only additions (or, equivalently, compute the power x^n by using only multiplications), then the above binary-based procedure is optimal.

This turned out to be only true for $n \le 14$. For $n = 15$, the binary procedure requires that we compute $2x$, $4x$, $8x$, and then compute $x + 2x + 4x + 8x$, to the total of 6 additions. However, we can compute $15x$ in only 5 additions: $2x = x + x$, $3x = x + 2x$, $6x = 3x + 3x$, $9x = 6x + 3x$, and $15x = 6x + 9x$; see, e.g., [7].

Fastest known methods: methods based on m-ary number representations. At present, the fastest known algorithms for multiplication via addition (or, equivalently, for fast multiplication) are based on the use of m-ary number representations for an appropriate m (not necessarily $m = 2$) [3, 7]. Specifically, once we have an m-ary representation

$$n = c_k \cdot m^k + c_{k-1} \cdot m^{k-1} + \ldots + c_1 \cdot m^1 + c_0,$$

we can compute $n \cdot x$ as follows:

`Compute` $2x = x + x$, $3x = 2x + x$, ..., $(m-1) \cdot x = ((m-2) \cdot x) + x$.
$a \leftarrow 0$

`for` $i = k$ `to` 0 `by` -1
  $a \leftarrow m \cdot a$
  $a \leftarrow a + (c_i \cdot x)$
`return` a.

Let us briefly explain this algorithm. At first, we take $a = 0$ and $i = k$. For this value i, we first get $a \leftarrow m \cdot 0 = 0$ and then $a \leftarrow 0 + c_k \cdot x$, so after this iteration, we get $a = c_k \cdot x$.

On the next iteration, we take $i = k - 1$. On this iteration, we first multiply the current value of a by m, resulting in $a = c_k \cdot m \cdot x$, and then add $c_{k-1} \cdot x$. So, after this iteration, we get $a = (c_k \cdot m + c_{k-1}) \cdot x$.

Similarly, after the next iteration corresponding to $i = k - 2$, we get $a = (c_k \cdot m^2 + c_{k-1} \cdot m + c_{k-2}) \cdot x$, ..., and after the last iteration corresponding to $i = 0$, we get the desired value $a = (c_k \cdot m^k + c_{k-1} \cdot m^{k-1} + \ldots + c_0) \cdot x = n \cdot x$.

Similarly, we can compute x^n as follows:

`Compute` $x^2 = x \cdot x$, $x^3 = x^2 \cdot x$, ..., $x^{m-1} = x^{m-2} \cdot x$.
$a \leftarrow 1$
`for` $i = k$ `to` 0 `by` -1
  $a \leftarrow a^m$
  $a \leftarrow a \cdot x^{c_i}$
`return` a.

Let us briefly explain this algorithm. At first, we take $a = 1$ and $i = k$. For this value i, we first get $a \leftarrow 1^m = 1$ and then $a \leftarrow 1 \cdot x^{c_k}$, so after this iteration, we get $a = x^{c_k}$.

On the next iteration, we take $i = k - 1$. On this iteration, we first raise the current value of a to the m-th power, resulting in $a = (x^{c_k})^m = x^{c_k \cdot m}$, and then multiply by $x^{c_{k-1}}$. So, after this iteration, we get $a = x^{c_k \cdot m + c_{k-1}}$.

Similarly, after the next iteration corresponding to $i = k - 2$, we get $a = x^{c_k \cdot m^2 + c_{k-1} \cdot m + c_{k-2}}$, ..., and after the last iteration corresponding to $i - 0$, we get the desired value

$$a = x^{c_k \cdot m^k + c_{k-1} \cdot m^{k-1} + \ldots + c_0} = x^n.$$

These methods is mainly used when $m = 2^p$, because then computing $m \cdot a$ requires only p additions (doublings) and, correspondingly, computing a^m requires only p multiplications (squarings).

Computational complexity (running time) of m-ary methods with $m = 2^p$. For $m = 2$, the above method requires $\lfloor \log_2(n) \rfloor$ doublings and $\le \lfloor \log_2(n) \rfloor$ additions. So, in the worst case, we need $2\lfloor \log_2(n) \rfloor$ additions.

In practice, if $c_i = 0$, then we do not need to add the corresponding value $2^i \cdot x$. On average, for each digit c_i, all m possible values $0, 1, \ldots, m - 1$ are equally probable. In particular, with the probability $1/m$, we get $c_i = 0$, in which case we do not need to add the corresponding term. For $m = 2$, this probability is $1/2$, so on average, we need $\lfloor \log_2(n) \rfloor$ doublings and $(1/2) \cdot \lfloor \log_2(n) \rfloor$ additions, to the overall of $(3/2) \cdot \lfloor \log_2(n) \rfloor$ additions.

For $m = 2^p$, we need $2^p - 2$ additions to compute $2x$, $3x$, ..., $(m - 1) \cdot x$, $\lfloor \log_2(n) \rfloor$ doublings (to compute a_m), and at most $\lfloor \log_2(n) \rfloor / p$ additions of $c_i \cdot x$. The overall worst-case complexity is thus $2^p - 2 + (1 + 1/p) \cdot \lfloor \log_2(n) \rfloor$ additions.

In the average case, we only need the addition of $c_i \cdot x$ when $c \ne 0$, i.e., with probability $1 - 1/m = 1 - 1/2^p$. Thus, the average-case complexity is equal to

$$2^p - 2 + \left(1 + \frac{1}{p} \cdot \left(1 - \frac{1}{2^p}\right)\right) \cdot \lfloor \log_2(n) \rfloor$$

additions [3].

It is known that we get the asymptotically fastest computations for

$$p = \log_2(\log_2(n)) - 2\log_2(\log_2(\log_2(n))).$$

When are methods with $m = 2$, $m = 4$, and $m = 8$ actually better? Analysis based on worst-case complexity. In some practical situations, it is important to guarantee that the computation finishes on time. In this case, it is desirable to minimize the worst-case complexity, because this is the complexity which provides the desired guarantee. Let us therefore compare the worst-case complexity t_p corresponding to different values $m = 2^p$.

For $p = 1$, we get $t_1 = 2\lfloor\log_2(n)\rfloor$. For $p = 2$, we get $t_2 = 2 + 1\frac{1}{2} \cdot \lfloor\log_2(n)\rfloor$. For $p = 3$, we get $t_3 = 6 + 1\frac{1}{3} \cdot \lfloor\log_2(n)\rfloor$.

The value $m = 2$ (corresponding to $p = 1$) is optimal when $t_1 \le t_2$, i.e., when $2\lfloor\log_2(n)\rfloor \le 2 + 1\frac{1}{2} \cdot \lfloor\log_2(n)\rfloor$. This is equivalent to $\frac{1}{2} \cdot \lfloor\log_2(n)\rfloor \le 2$, i.e., to $\lfloor\log_2(n)\rfloor \le 4$ and $n < 2^5 = 32$.

The value $m = 4$ (corresponding to $p = 2$) is optimal when $t_1 > t_2$ (i.e., when $n \ge 32$) and $t_2 \le t_3$, i.e., when $2 + 1\frac{1}{2} \cdot \lfloor\log_2(n)\rfloor \le 6 + 1\frac{1}{3} \cdot \lfloor\log_2(n)\rfloor$. This condition is equivalent to $\frac{1}{6} \cdot \lfloor\log_2(n)\rfloor \le 4$, i.e., to $\lfloor\log_2(n)\rfloor \le 24$ and $n < 2^{25} \approx 3 \cdot 10^7$.

Thus, for the values n which do not exceed 30 million (i.e., in practice, in all practical cases when we need estimates), the granularity values of $m = 2$ and $m = 4$ are optimal – and $m = 2$ is only optimal for small values n, when we do not really need any estimation. Crudely speaking, we can say that the worst-case complexity corresponds to $m = 4$.

When are methods with $m = 2$, $m = 4$, and $m = 8$ actually better? Analysis based on average-case complexity. In some practical situations, we need to perform several computations, with several different values x; in some such situations, the individual computation time is not crucial, what is important is that the overall computation time be as small as possible. In such situations, it makes sense to consider the average time complexity $\bar{t}_p$ as an optimality criterion.

For $p = 1$, we get $\bar{t}_1 = \frac{3}{2} \cdot \lfloor\log_2(n)\rfloor$. For $p = 2$, we get

$$\frac{1}{2} \cdot \left(1 - \frac{1}{4}\right) = \frac{1}{2} \cdot \frac{3}{4} = \frac{3}{8},$$

so $\bar{t}_2 = 2 + 1\frac{3}{8} \cdot \lfloor\log_2(n)\rfloor$. For $p = 3$, we get

$$\frac{1}{3} \cdot \left(1 - \frac{1}{8}\right) = \frac{1}{3} \cdot \frac{7}{8} = \frac{7}{24},$$

so $\bar{t}_3 = 6 + 1\frac{7}{24} \cdot \lfloor\log_2(n)\rfloor$.

In this case, the granularity value $m = 2$ corresponding to $p = 1$ is optimal when $\bar{t}_1 \le \bar{t}_2$, i.e., when $\frac{3}{2} \cdot \lfloor\log_2(n)\rfloor \le 2 + 1\frac{3}{8} \cdot \lfloor\log_2(n)\rfloor$. This condition is equivalent to $\left(\frac{1}{2} - \frac{3}{7}\right) \cdot \lfloor\log_2(n)\rfloor \le 2$, i.e., to $\frac{1}{14} \cdot \lfloor\log_2(n)\rfloor \le 2$, $\lfloor\log_2(n)\rfloor \le 28$, and $n < 2^{29} \approx 5 \cdot 10^8$. Thus, for all practical values, the granularity value $m = 2$ is optimal.

Conclusion. J. Hobbs has observed that for human experts, it is natural to express their rough estimates in terms of half-orders of magnitude (HOM), when there are approximately two possible estimates within each order of magnitude (i.e., within each factor of 10). For example, when estimating a size of a crowd, a human naturally distinguishes between "low hundreds", "high hundreds", "low thousands", "high thousands", etc. How can we explain this granule size?

In this paper, we show that for values appropriate for human estimation, from the viewpoint of data processing under refined granularity, the optimal granule size is either $m = 4$ (for the more typical case of individual problems), or $m = 2$ (for mass problems). In both cases, we have a granule size which is similar to half-order of magnitude. So, we get a new theoretical explanation for the HOM phenomenon observed by J. Hobbs.

References

[1] L. N. H. Bunt, P. S. Jones, and J. D. Bedient, *The Historical Roots of Elementary Mathematics*, Dover, New York, 1988.

[2] Th. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 2001.

[3] D. M. Gordon, "A survey of fast exponentiation methods", *Journal of Algorithms*, 1998, Vol. 27, pp. 129–146.

[4] J. R. Hobbs, "Half orders of magnitude", In: L. Obrst and I. Mani (eds.), *Proceeding of the Workshop on Semantic Approximation, Granularity, and Vagueness, A Workshop of the Seventh International Conference on Principles of Knowledge Representation and Reasoning KR'2000*, Breckenridge, Colorado, April 11, 2000, pp. 28–38.

[5] J. R. Hobbs and V. Kreinovich, "Optimal Choice of Granularity In Commonsense Estimation: Why Half-Orders of Magnitude", *Proceedings of the Joint 9th World Congress of the International Fuzzy Systems Association and 20th International Conference of the North American Fuzzy Information Processing Society IFSA/NAFIPS 2001*, Vancouver, Canada, July 25–28, 2001, pp. 1343–1348.

[6] J. Hobbs and V. Kreinovich, "Optimal Choice of Granularity In Commonsense Estimation: Why Half-Orders of Magnitude", *International Journal of Intelligent Systems*, 2006, Vol. 21, No. 8, pp. 843–855.

[7] D. E. Knuth, *The Art of Computer Programming. Vol. 2. Seminumerical Algorithms*, Addison Wesley, Reading, Massachusetts, 1969.