The changing $\times$: Multiplication algorithms, new and old

Ricky Liu

University of Washington Math Hour

May 22, 2022

Addition is easy.

$$
\begin{array}{r}
{\scriptstyle 1}{\scriptstyle 1}{\scriptstyle 1}{\scriptstyle 1} \\
2\,3\,8\,4\,7\,6\,2\,4\,5\,1 \\
+\quad 8\,4\,2\,1\,7\,9\,3\,6\,2\,7 \\
\hline
1\,0\,8\,0\,6\,5\,5\,6\,0\,7\,8
\end{array}
$$

Multiplication is hard.

$$
\begin{array}{r}
2384762451 \\
\times\, 8421793627 \\
\hline
16693337157 \\
4769524902 \\
14308574706 \\
7154287353 \\
21462862059 \\
16693337157 \\
2384762451 \\
4769524902 \\
9539049804 \\
19078099608 \\
\hline
20083977211740699777
\end{array}
$$

Question

What is the fastest way to multiply?

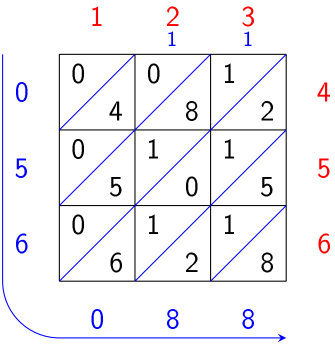
The Standard Algorithm

$$
\begin{array}{r}
1\;2\;3 \\
\times\;4\;5\;6 \\
\hline
7\;3\;8 \\
6\;1\;5\; \\
4\;9\;2\;\; \\
\hline
5\;6\;0\;8\;8
\end{array}
$$

Requires:

- multiplying every digit in the first number by every digit in the second number;
- knowledge of a 10×10 multiplication table.

The lattice or grid method:



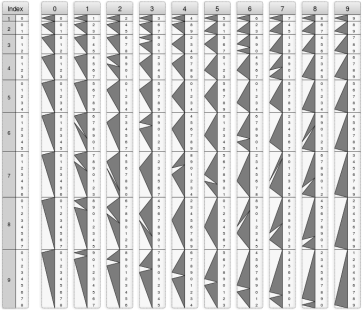
$$123 \times 456 = 56088$$

The underlying process is the same as the standard algorithm (the same multiplications and additions are done but in a slightly different order).

The lattice method was used in various historical computing devices.

Napier's bones (1617)

Genaille-Lucas rulers (1891)



Peasant multiplication

What if you don't have a multiplication table memorized?
Enter Russian peasant multiplication, based on doubling and halving.

odd

$$\begin{array}{rr}
41 & \times\ 23 \\
\cancel{20} & \cancel{46} \\
\cancel{10} & \cancel{92} \\
5 & 184 \\
\cancel{2} & \cancel{368} \\
1 & 736 \\
\hline
 & 943
\end{array}$$

Requires:

- Knowledge of addition and halving;
- More steps than the standard algorithm, but the steps are simpler.

A similar method involving only doubling was used by the ancient Egyptians.

$$41 \times 23 = (32 + 8 + 1) \times 23$$

$$\begin{array}{rr}
1 & 23 \\
\cancel{2} & \cancel{46} \\
\cancel{4} & \cancel{92} \\
8 & 184 \\
\cancel{16} & \cancel{368} \\
32 & 736 \\
\hline
41 & 943
\end{array}$$

Actually, this is essentially the standard algorithm in binary!

$$\begin{array}{rrrrrrrrrr} & & & & & 1 & 0 & 1 & 1 & 1 \\ \times & & & & 1 & 0 & 1 & 0 & 0 & 1 \\ \hline & & & & & 1 & 0 & 1 & 1 & 1 \\ & & 1 & 0 & 1 & 1 & 1 & & & \\ 1 & 0 & 1 & 1 & 1 & & & & & \\ \hline 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \end{array} \qquad \begin{array}{lr} & 23 \\ \times & 41 \\ \hline & 23 \\ & 184 \\ & 736 \\ \hline & 943 \end{array}$$

Table-based methods

Instead of a multiplication table, some methods used other tables.

1	0
2	1
3	2
4	4
5	6

6	9
7	12
8	16
9	20
10	25

11	30
12	36
13	42
14	49
15	56

16	64
17	72
18	81
19	90
20	100

...

156	6084
157	6162
158	6241
159	6320
160	6400

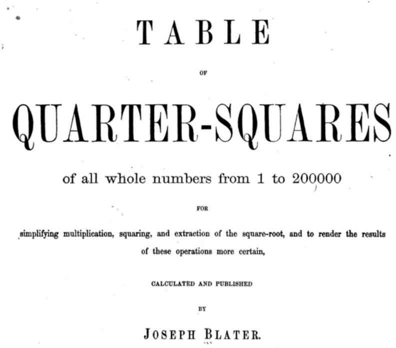
To multiply 83×74:

$$\begin{array}{lcr} 83 + 74 = 157 & \rightarrow & 6162 \\ 83 - 74 = 9 & \rightarrow & 20 \\ \hline & & 6142 \end{array}$$

This is the Babylonian quarter-square method. It uses the identity

$$\frac{(x+y)^2}{4} - \frac{(x-y)^2}{4} = xy.$$

To multiply numbers up to n, you need $2n$ quarter-squares (as opposed to n^2 entries in a multiplication table).



TABLE

OF

QUARTER-SQUARES

of all whole numbers from 1 to 200000

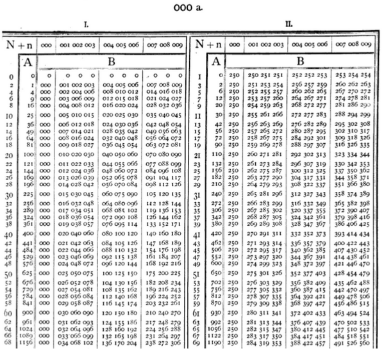
FOR

simplifying multiplication, squaring, and extraction of the square-root, and to render the results of these operations more certain,

CALCULATED AND PUBLISHED

BY

JOSEPH BLATER.



000 a.

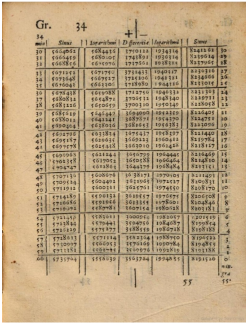
I.

N + n	A	000	001 002 003	004 005 006	007 008 009
	A	B			
0	0	0	0 0 0	0 0 0	0 0 0
2	1	000	001 002 003	004 005 006	007 008 009
4	4	000	002 004 006	008 010 012	014 016 018
6	9	000	003 006 009	012 015 018	021 024 027
8	16	000	004 008 012	016 020 024	028 032 036
10	25	000	005 010 015	020 025 030	035 040 045
12	36	000	006 012 018	024 030 036	042 048 054
14	49	000	007 014 021	028 035 042	049 056 063
16	64	000	008 016 024	032 040 048	056 064 072
18	81	000	009 018 027	036 045 054	063 072 081
20	100	000	010 020 030	040 050 060	070 080 090
22	121	000	011 022 033	044 055 066	077 088 099
24	144	000	012 024 036	048 060 072	084 096 108
26	169	000	013 026 039	052 065 078	091 104 117
28	196	000	014 028 042	056 070 084	098 112 126
30	225	000	015 030 045	060 075 090	105 120 135
32	256	000	016 032 048	064 080 096	112 128 144
34	289	000	017 034 051	068 085 102	119 136 153
36	324	000	018 036 054	072 090 108	126 144 162
38	361	000	019 038 057	076 095 114	133 152 171
40	400	000	020 040 060	080 100 120	140 160 180
42	441	000	021 042 063	084 105 126	147 168 189
44	484	000	022 044 066	088 110 132	154 176 198
46	529	000	023 046 069	092 115 138	161 184 207
48	576	000	024 048 072	096 120 144	168 192 216
50	625	000	025 050 075	100 125 150	175 200 225
52	676	000	026 052 078	104 130 156	182 208 234
54	729	000	027 054 081	108 135 162	189 216 243
56	784	000	028 056 084	112 140 168	196 224 252
58	841	000	029 058 087	116 145 174	203 232 261
60	900	000	030 060 090	120 150 180	210 240 270
62	961	000	031 062 093	124 155 186	217 248 279
64	1024	000	032 064 096	128 160 192	224 256 288
66	1089	000	033 066 099	132 165 198	231 264 297
68	1156	000	034 068 102	136 170 204	238 272 306

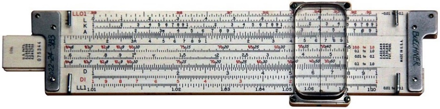
II.

N + n	A	000	001 002 003	004 005 006	007 008 009
	A	B			
1	0	250	250 251 251	252 252 253	253 254 254
3	2	250	251 253 254	256 257 259	260 262 263
5	6	250	252 255 257	260 262 265	267 270 272
7	12	250	253 257 260	264 267 271	274 278 281
9	20	250	254 259 263	268 272 277	281 286 290
11	30	250	255 261 266	272 277 283	288 294 299
13	42	250	256 263 269	276 282 289	295 302 308
15	56	250	257 265 272	280 287 295	302 310 317
17	72	250	258 267 275	284 292 301	309 318 326
19	90	250	259 269 278	288 297 307	316 326 335
21	110	250	260 271 281	292 302 313	323 334 344
23	132	250	261 273 284	296 307 319	330 342 353
25	156	250	262 275 287	300 312 325	337 350 362
27	182	250	263 277 290	304 317 331	344 358 371
29	210	250	264 279 293	308 322 337	351 366 380
31	240	250	265 281 296	312 327 343	358 374 389
33	272	250	266 283 299	316 332 349	365 382 398
35	306	250	267 285 302	320 337 355	372 390 407
37	342	250	268 287 305	324 342 361	379 398 416
39	380	250	269 289 308	328 347 367	386 406 425
41	420	250	270 291 311	332 352 373	393 414 434
43	462	250	271 293 314	336 357 379	400 422 443
45	506	250	272 295 317	340 362 385	407 430 452
47	552	250	273 297 320	344 367 391	414 438 461
49	600	250	274 299 323	348 372 397	421 446 470
51	650	250	275 301 326	352 377 403	428 454 479
53	702	250	276 303 329	356 382 409	435 462 488
55	756	250	277 305 332	360 387 415	442 470 497
57	812	250	278 307 335	364 392 421	449 478 506
59	870	250	279 309 338	368 397 427	456 486 515
61	930	250	280 311 341	372 402 433	463 494 524
63	992	250	281 313 344	376 407 439	470 502 533
65	1056	250	282 315 347	380 412 445	477 510 542
67	1122	250	283 317 350	384 417 451	484 518 551
69	1190	250	284 319 353	388 422 457	491 526 560

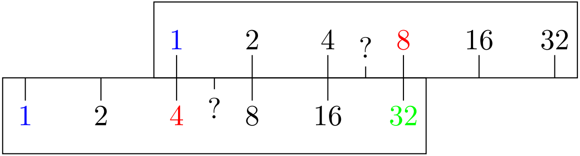
A different table-based method was introduced by John Napier in 1614.



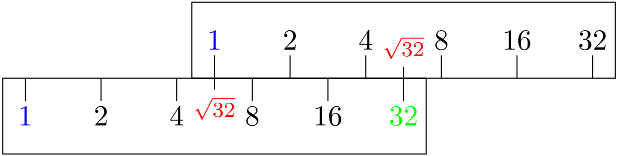
It was turned into a computing device by William Oughtred in 1622.



A simple slide rule:



What number should go here?



$$\sqrt{32} = 2^{2.5} \approx 5.657$$

In general, the number located d units from the left is 2^d.

If $x = 2^d$, then $d = \log_2 x$ is the (base-2) logarithm of x.

With a table of logarithms, you can do multiplication with just addition and a few lookups.

x	$\log_{10} x$
1.0	.00000
1.1	.04139
1.2	.07918
1.3	.11394
1.4	.14613
1.5	.17609
1.6	.20412
1.7	.23045
1.8	.25527
1.9	.27875
2.0	.30103

$$\begin{array}{rrlcrr} & 13 & = 1.3 \times 10^1 & \rightarrow & & 1.11394 \\ \times & 14 & = 1.4 \times 10^1 & \rightarrow & + & 1.14613 \\ \hline & \approx 180 & = 1.8 \times 10^2 & \leftarrow & & 2.26007 \end{array}$$

It can be a bit imprecise...

Computational complexity

Question

What is the fastest way to multiply?

Question

How do we judge the speed of an algorithm?

A: Count the number of operations required with n digit numbers as inputs.

For example, adding two n digit numbers requires n one-digit additions and potentially n carries, for a total of $2n$ operations.

What about multiplication?

For the standard method:

- n^2 one-digit multiplications,
- $\approx n^2$ 2-digit additions (equivalent to $\approx 2n^2$ one-digit additions)

for a total of about $3n^2$ operations.

For peasant multiplication:

- There are $\approx 3.3n$ rows.
- For each row, we may have to do an $\approx n$-digit halving, doubling, and addition ($\approx 3n$ operations),

for a total of about $10n^2$ operations.

We say that both algorithms run in $O(n^2)$ operations.

Big O notation

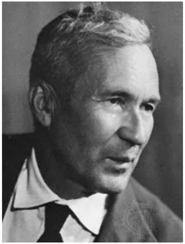
O is 'Big O' notation meaning roughly, "on the order of" or "up to a constant factor." Thus $O(n)$ could mean $2n$ or $999n + 7$.

Why don't we care about constant factors?

For really really big n, the constant is not important: any $O(n)$ algorithm will be faster than any $O(n^2)$ algorithm for $n \gg 0$, even if it is slower for small n due to more "overhead."

For instance, $999n < n^2$ when $n > 999$.

Kolmogorov's conjecture



In 1960, Russian mathematician Andrey Kolmogorov made the following conjecture at a conference.

Conjecture

Any algorithm to multiply two n-digit numbers requires at least $O(n^2)$ steps.

This conjecture intrigued 23-year-old student Anatoly Karatsuba.

Within a week, Karatsuba had disproved the conjecture by finding a way to multiply two n-digit numbers using $O(n^{1.58})$ operations!

Kolmogorov was so pleased by the result that he wrote it up and had it published on Karatsuba's behalf.

Karatsuba multiplication

Consider multiplying two-digit numbers using the standard method (before performing carries).

$$
\begin{array}{ccc}
 & a & b \\
\times & c & d \\
\hline
 & a \times d & b \times d \\
a \times c & b \times c & \\
\hline
a \times c & (a \times d) + (b \times c) & b \times d
\end{array}
$$

To multiply $ab \times cd$, we need to find:

- $X = a \times c$
- $Y = b \times d$
- $Z = (a \times d) + (b \times c)$

It seems like we need to do 4 multiplications.

But there is another way: note that

$$\begin{aligned}(a+b) \times (c+d) &= (a \times c) + (a \times d) + (b \times c) + (b \times d) \\ &= X + Z + Y\end{aligned}$$

Thus

$$Z = (a+b) \times (c+d) - X - Y.$$

Then we can find X, Y, and Z using only 3 multiplications instead of 4 at the expense of more additions/subtractions.

$$
\begin{array}{cccc}
 & & 5 & 3 \\
\times & & 2 & 7 \\
\hline
 & & 2 & 1 \\
 & 4 & 1 & \\
1 & 0 & & \\
\hline
1 & 4 & 3 & 1
\end{array}
$$

$$
\begin{aligned}
X &= 5 \times 2 = 10 \\
Y &= 3 \times 7 = 21 \\
Z &= (5+3) \times (2+7) - X - Y \\
&= 8 \times 9 - 10 - 21 = 41
\end{aligned}
$$

We traded a multiplication for a bunch of additions. Is this really faster?

Not for two-digit numbers...

But we can also use this idea for numbers with more digits!

$$
\begin{array}{rrrr}
 & & 3825 & 4926 \\
\times & & 2937 & 6328 \\
\hline
 & & 3117 & 1728 \\
 & ???? & ???? & \\
1123 & 4025 & & \\
\hline
\end{array}
$$

(standard)

$$Z = 3825 \times 6328 + 4926 \times 2937$$

(Karatsuba)

$$Z = (3825 + 4926) \times (2937 + 6328) \\ - 31171728 - 11234025$$

We're replacing a hard multiplication with easy additions/subtractions, which are much faster!

We can divide and conquer to get more savings by using Karatsuba's algorithm for the smaller multiplications.

To multiply two 16-digit numbers, Karatsuba would do:

- 1 16-digit multiplication $\to$ 3 8-digit multiplications
- 3 8-digit multiplications $\to$ 9 4-digit multiplications
- 9 4-digit multiplications $\to$ 27 2-digit multiplications
- 27 2-digit multiplications $\to$ 81 1-digit multiplications

Compare with $16^2 = 256$ 1-digit multiplications for the standard algorithm.

For 1000-digit numbers, the standard algorithm needs 1000000 1-digit multiplications while Karatsuba needs only 60000.

In general, Karatsuba's algorithm uses only

$$O(n^{\log_2 3}) \approx O(n^{1.58})$$

operations.

Can we do better?

- Karatsuba (1960) $O(n^{1.58})$
  - Can multiply 2-digit numbers with 3 multiplications instead of 4
- Toom-Cook (1963) $O(n^{1.46})$
  - Can multiply 3-digit numbers with 5 multiplications instead of 9
  - Can make 1.46 close to 1 with more pieces but a lot of overhead
- Schönhage and Strassen (1971) $O(n \cdot \log n \cdot \log\log n)$
  - Based on Fast Fourier Transform
  - Faster for numbers > 10000 digits
- Fürer (2007) $O(n \cdot \log n \cdot 2^{O(\log^* n)})$
  - Slower for practical applications due to large overhead
- Harvey and van der Hoeven (2021) $O(n \cdot \log n)$

Open question

Is there an algorithm for multiplying n-digit numbers that is faster than $O(n \cdot \log n)$?