The Euclidean Algorithm and Bezout's Identity

Matt Draisey

Number Theory

In it's simplest form Bezout's Identity is a statement of the fact that the greatest common divisor of any two numbers can be written as a linear combination of them. Bezout's Identity, the Euclidean Algorithm and the Chinese Remainder Theorem (a trick in number theory whereby a number is reconstructed given its remainder with respect to different bases) are core concepts in number theory and should be very familiar to any amateur mathematician. It's hard to imagine how anyone could take thirteen pages in explaining the mechanics of how to calculate the coefficients of Bezout's Identity, especially as we will not explore any techniques outside of the ordinary one's of the Euclidean Algorithm.

But even though the technique is simple and effective, it does present just enough subtlety to warrant this exploration. And the development of the algorithms will give us different constructive proofs of Bezout's Identity which are interesting in of themselves.

Theorem as Succinct Python Code: Bezout's Identity

Let all variables be nonzero natural numbers. For $a, b \neq d = \gcd(a, b)$ there is a unique mapping $(a, b) \mapsto (mm, nn), (m, n)$ with $mm, m < {}^{a}\!/_{d}$ and $nn, n < {}^{b}\!/_{d}$ such that $\left|\begin{smallmatrix} mm & nn \\ a & b \end{smallmatrix}\right| = d = \left|\begin{smallmatrix} a & b \\ m & n \end{smallmatrix}\right|$

```python
def bezouts_mapping(a, b, /):
    if b != 0:
        q, r = divmod(a, b)
        (mm, nn), (m, n) = bezouts_mapping(b, r)
        return (q*m + n, m), (q*mm + nn, mm)
    else:
        return (1, -1), (0, 1)
```

Note that this mapping code is an adjunct to the Euclidean algorithm and could easily have passed back the greatest common divisor as revealed in the deepest level of the recursion.

Exploration

We will explore both the strictest and the loosest forms of Bezout's Identity for natural numbers. We wont prove uniqueness (it's easy to show given Bezout's Identity without uniqueness) nor will we explore extensions to the integers or other domains. What we will show is four quite distinct variations on the basic algorithm with code examples in Python. Each of these is a built around the Euclidean Algorithm and makes no attempt to improve upon this foundation.

Our first algorithm is that implicit to the most straightforward general inductive proof of Bezout's Identity. The functionally pure code example to go along with it is the most elegant but not necessarily the best.

Theorem via the Euclidean Algorithm: Bezout's Identity Let a and b be natural numbers distinct from their greatest common divisor d. Then there are nonzero natural numbers $m^+ < {}^a/_d$ and $n^+ < {}^b/_d$ such that $m^+b - n^+a = d$ and nonzero natural numbers $m^- < {}^a/_d$ and $n^- < {}^b/_d$ such that $n^-a - m^-b = d$

$$\begin{vmatrix} m^\pm & a \\ n^\pm & b \end{vmatrix} = \pm d \qquad \text{or, within the natural numbers,} \qquad \begin{vmatrix} m^+ & a \\ n^+ & b \end{vmatrix} = d = \begin{vmatrix} a & m^- \\ b & n^- \end{vmatrix}$$

General Inductive Hypothesis Let a, b and d be as above. Then a and b must be distinct, nonzero and greater than d. So require $a > b > d$. The general inductive hypothesis is that the lemma already holds for any such pair of natural numbers with the same gcd and strictly less that a.

Proof of the Inductive Step By long division of the natural number $b > 0$ into the natural number a

$$a = qb + r \text{ where natural number } q \leqslant a \text{ and natural number } r < b$$

Division of the smaller b into the larger a implies that $q > 0$; that $b > d$ further implies that $q < {}^a/_d$.

That d divides a and d divides b implies d divides $r = a - qb$, i.e. r must be of the form 0, d, $2d$, $3d$, ... The assumption that b is greater than the greatest common divisor of a and b implies that b is not itself a divisor of a. So r is not zero and must be a positive multple of d.

When‡ $r = d$ we solve directly for the $Mb - Na = \pm d$ constructions ($+d$ and $-d$ respectively).
For the latter construction we define nonzero natural numbers $M = q < {}^a/_d$ and $N = 1 < {}^b/_d$ so that

$$\begin{vmatrix} M & a \\ N & b \end{vmatrix} = \begin{vmatrix} q & a \\ 1 & b \end{vmatrix} = qb - a = -r = -d \tag{*}$$

For the former we define nonzero natural numbers $M = {}^a/_d - q < {}^a/_d$ and $N = {}^b/_d - 1 < {}^b/_d$ so that

$$\begin{vmatrix} M & a \\ N & b \end{vmatrix} = \begin{vmatrix} {}^a/_d - q & a \\ {}^b/_d - 1 & b \end{vmatrix} = \cancel{\begin{vmatrix} a & a \\ b & b \end{vmatrix} \frac{1}{d}} - \begin{vmatrix} q & a \\ 1 & b \end{vmatrix} = r = +d \tag{**}$$

When‡‡ $r > d$ we have $a > b > r > d$ where d divides a, b and r. And were b and r to have a common divisor greater than d then so would $a = qb + r$ which contradicts the assumption that d is the greatest common divisor of a and b. So the greatest common divisor of b and r is d and we can invoke the inductive hypothesis on $b > r > d$. So there are m and n with $m < {}^b/_d$ and $n < {}^r/_d$ such that $mr - nb = \pm d$ (collapsing the two constructions into one)

$$\begin{aligned} a - qb &= r \\ ma - mqb &= mr \\ ma - mqb &= nb \pm d \\ ma - (qm + n)b &= \pm d \\ (qm + n)b - ma &= \mp d \quad (\mp \text{ denotes a flipped the } \pm \text{ sign}) \end{aligned} \tag{†}$$

We define $M = qm + n$ and $N = m$ (criss-crossing the two constructions $M^\mp = qm^\pm + n^\pm$, $N^\mp = m^\pm$)

$$\begin{matrix} m < {}^b/_d \\ qm < {}^{qb}/_d \end{matrix} \quad \text{and} \quad n < {}^r/_d \quad \Longrightarrow \quad \begin{matrix} M = qm + n < qm + {}^r/_d \\ \wedge \qquad\qquad \wedge \\ {}^{qb}/_d + n < {}^{qb}/_d + {}^r/_d = {}^{(qb+r)}/_d = {}^a/_d \end{matrix} \tag{††}$$

To get nonzero $M < {}^a/_d$ and $N < {}^b/_d$ such that

$$\begin{vmatrix} M & a \\ N & b \end{vmatrix} = \mp d$$

By Induction The theorem is true for any pair of natural numbers distinct from their gcd. □

Generalization: via ‡Elaboration on the Terminal $r = d$ Case of the Inductive Proof When

$$a = qb + r \text{ where natural number } 0 < q < {}^{a}\!/_{d} \text{ and natural number } r = d < b$$

In the inductive proof we simply present the M and N coefficients for the terminal $r = d$ case. To better understand this choice of $M = n_0$ and $N = n_1$ we will regress one generation using the relevant iterative step (†) for the coefficents n_1 and n_2 and then a further generation of the iteration beyond that — as would be necessary for an actual implementation of the algorithm to detect the end of recursion

$$\begin{array}{ccc} (q, d) = a \operatorname{divMod} b & \multirow{2}{*}{$\rightarrow$} & ({}^{b}\!/_{d}, 0) = b \operatorname{divMod} d \\ a = qb + d & & b = {}^{b}\!/_{d} \cdot d + 0 \end{array}$$

for the (*) construction

$$\begin{bmatrix} q \\ 1 \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} q & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} b/d & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$n_0 = qn_1 + n_2 \quad \leftarrow \quad n_1 = {}^{b}\!/_{d} \cdot n_2 + n_3$$

$$\begin{bmatrix} n_0 \\ n_1 \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} q & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} n_1 \\ n_2 \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} b/d & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} n_2 \\ n_3 \end{bmatrix}$$

$$\begin{vmatrix} n_0 & a \\ n_1 & b \end{vmatrix} = \mp d \qquad \begin{vmatrix} n_1 & b \\ n_2 & d \end{vmatrix} = \pm d \qquad \begin{vmatrix} n_2 & d \\ n_3 & 0 \end{vmatrix} = \mp d$$

for the (**) construction

$$\begin{bmatrix} {}^{a}\!/_{d} - q \\ {}^{b}\!/_{d} - 1 \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} q & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} {}^{b}\!/_{d} - 1 \\ 1 \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} b/d & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

noting that ${}^{a}\!/_{d} = {}^{(qb+d)}\!/_{d} = q \cdot {}^{b}\!/_{d} + 1$

The 2×2 transformation matrices $\left[\begin{smallmatrix} q & 1 \\ 1 & 0 \end{smallmatrix}\right]$ or $\left[\begin{smallmatrix} b/d & 1 \\ 1 & 0 \end{smallmatrix}\right]$ associated with the quotients q or ${}^{b}\!/_{d}$ are a convenient way to denote the iterative step on the paired state $\left[\begin{smallmatrix} n_0 \\ n_1 \end{smallmatrix}\right] \leftarrow \left[\begin{smallmatrix} n_1 \\ n_2 \end{smallmatrix}\right]$ or $\left[\begin{smallmatrix} n_1 \\ n_2 \end{smallmatrix}\right] \leftarrow \left[\begin{smallmatrix} n_2 \\ n_3 \end{smallmatrix}\right]$. Each generation of the algorithm only generates a single new n_j value; a shifting window of two successive elements from this sequence of n_j gives us the pair of coefficients for Bezout's Identity. This mirrors the way the simple Euclidean Algorithm, to compute the gcd, maintains a state of two values. It will convenient for us to list the more complex (**) construction to the left of the simpler (*) one

	n^{***}				n^{**}	n^*		0
a	${}^{a}\!/_{d}$	$>$	n_0	$=$	${}^{a}\!/_{d} - q$	q	$>$	0
b	${}^{b}\!/_{d}$	$>$	n_1	$=$	${}^{b}\!/_{d} - 1$	1	$>$	0
d	1	$\geqslant$	n_2	$=$	1	0	$\geqslant$	0
0	0		n_3	$=$	-1	1		0

where the potential upper bound $n^{***} = n^{**} + n^*$

Consider the limits on the n_j. The identity $n_2 0 - n_3 d = \mp d$ implies that $b_3 = \pm 1$. Note that $n_3^{**} = -1$ is the only exception to the rule that the n_j are natural numbers and $n_2^* = 0$ is the only n_j which can be zero — all other n_j are nonzero natural numbers. The $n_3 = \pm 1$ have no particular bound but all the other n_j are bounded — and here $n_2^{**} = 1$ is the only exception to the rule that the n_j are bounded above by a strict inequality and may not reach their bound.

The (††) statement of the inductive proof ensures that, no matter how many iterations of the recursive algorithm implicit in the general recursive proof, the strict bounds on the Bezout's Identity coefficients will hold for any pair of natural numbers $a, b > \gcd(a, b)$. The only way one of the looser bounds could hold would be if no recursion occurs at all — i,e, the first iteration of the algorithm were to return a zero remainder. If we were to loosen the preconditions of the theorem so that natural numbers b and d be a candidate pair where $b > d = \gcd(b, d)$ then

	n^{***}				n^{**}	n^*		
b	${}^{b}\!/_{d}$	$>$	n_1	$=$	${}^{b}\!/_{d} - 1$	1	$>$	0
d	1	$\geqslant$	n_2	$=$	1	0	$\geqslant$	0
0	0		n_3	$=$	-1	1		0

$$\begin{vmatrix} n_1^{**} & b \\ n_2^{**} & d \end{vmatrix} = \begin{vmatrix} {}^{b}\!/_{d} - 1 & b \\ 1 & d \end{vmatrix} = -d \qquad \begin{vmatrix} n_1^{*} & b \\ n_2^{*} & d \end{vmatrix} = \begin{vmatrix} 1 & b \\ 0 & d \end{vmatrix} = d$$

If we were further loosen the preconditions of the theorem so that $b = d = \gcd(b, d)$ then we would need to loosen the strict bounds a little bit more so that

	n^{***}				n^{**}	n^*		
b	1	$\geqslant$	n_1	$=$	0	1	$\geqslant$	0
d	1	$\geqslant$	n_2	$=$	1	0	$\geqslant$	0
0	0		n_3	$=$	-1	1		0

$$\begin{vmatrix} n_1^{**} & b \\ n_2^{**} & d \end{vmatrix} = \begin{vmatrix} 0 & b \\ 1 & d \end{vmatrix} = -b = -d \qquad \begin{vmatrix} n_1^{*} & b \\ n_2^{*} & d \end{vmatrix} = \begin{vmatrix} 1 & b \\ 0 & d \end{vmatrix} = d$$

And should the preconditions be such that one of the pair of given numbers is zero the Bezout coefficients would not be wholly determined. Nevertheless the implied algorithm would return specific values that are a little strange but do satisfy the requirements (even for the problematic case when both givens are zero)

$$\begin{array}{cccccrrcc} \multicolumn{2}{c}{n^{***}} & & & & n^{**} & n^{*} & & \\ \hline d & 1 & \geqslant & n_2 & = & 1 & 0 & \geqslant & 0 \\ 0 & 0 & & n_3 & = & -1 & 1 & & 0 \end{array} \qquad \begin{vmatrix} n_2^{**} & d \\ n_3^{**} & 0 \end{vmatrix} = \begin{vmatrix} 1 & d \\ -1 & 0 \end{vmatrix} = d \qquad \begin{vmatrix} n_2^{*} & d \\ n_3^{*} & 0 \end{vmatrix} = \begin{vmatrix} 0 & d \\ 1 & 0 \end{vmatrix} = -d$$

The $\gcd(0,0)$ isn't necessarily well-defined but is usually considered to be zero, as the Euclidean Algorithm would determine. However, it may be more idiomatic to add special handling for these coefficients

$$\begin{array}{cccccrrc} \multicolumn{2}{c}{n^{***}} & & & & n^{**} & n^{*} & \\ \hline 0 & 0 & & n_3 & = & -1 & 1 & 0 \\ 0 & 0 & & n_4 & = & 1 & -1 & 0 \end{array} \qquad \begin{vmatrix} n_3^{**} & 0 \\ n_4^{**} & 0 \end{vmatrix} = \begin{vmatrix} -1 & 0 \\ 1 & 0 \end{vmatrix} = 0 \qquad \begin{vmatrix} n_3^{*} & 0 \\ n_4^{*} & 0 \end{vmatrix} = \begin{vmatrix} 1 & 0 \\ -1 & 0 \end{vmatrix} = 0$$

There is one potential weakness of the generated coefficients in the former case without special handling. We can express any ratio in least terms with the n^{***}, something we haven't proven but isn't terribly surprising. In the problematic $0:0$ case using the former coefficients $n_2^{***} : n_3^{***}$

$$d : 0 \overset{\text{in proportion}}{=} 1 : 0 \text{ (in least terms)} \overset{d=0}{\Longrightarrow} 0 : 0 \overset{\text{in proportion}}{=} 1 : 0 \quad ↯ \text{ a contradiction}$$

which is annoying whereas with the latter coefficients $n_3^{***} : n_4^{***}$ we get the correct

$$0 : 0 \overset{\text{in proportion}}{=} 0 : 0 \text{ (in least terms)} \checkmark$$

By recognizing this special case we can express any ratio in least terms. But we could ignore this final case in our code examples as it adds complication (a conditional in 2 lines of code) for very little benefit.

We have seen that by relaxing the preconditions of the inductive proof the implied algorithm still returns sensible results with post conditions that are only marginally weaker. Statement of the proof to go along with this algorithm is a little involved but completely straightforward. □

A Recursive Implementation of the General Inductive Algorithm in Python Note the algorithm as implied by the proof requires its second argument to be smaller than its first. This is easy to arrange but actually unneeded as, when given an a and b such that $a < b$ the first division tried by the algorithm will, having returned a zero quotient, swap the a and b as its first iterative step.

# $(a,b) \mapsto d, (a/d, b/d), (mm, nn), (m, n)$ *where* d *is* $\gcd(a,b) = \left|\begin{smallmatrix} mm & nn \\ a & b \end{smallmatrix}\right| = \left|\begin{smallmatrix} a & b \\ m & n \end{smallmatrix}\right|$

```
def euclidean_bezout_1(a, b, /):
    if a == 0 and b == 0:
        return 0, (0, 0), (-1, 1), (1, -1)
    else
        return recursive_euclidean_bezout(a, b)

def recursive_euclidean_bezout(a, b, /):
    if b != 0:
        q, r = divmod(a, b)
        d, (mmm, nnn), (mm, nn), (m, n) = recursive_euclidean_bezout(b, r)
        return d, (q*mmm + nnn, mmm), (q*m + n, m), (q*mm + nn, mm)
    else:
        return a, (1, 0), (1, -1), (0, 1)
```

That's a really short piece of code for such a long wind-up. Note that the code is purely functional and does not involve mutable state — such is the character of code directly derived from an inductive proof. And unlike most implementations of the Euclidean Algorithm which only return the gcd, this code is not tail-recursive and it cannot be rewritten as a while loop — at least not as stated. ◇

Corollary: A Two-Phase Algorithm via Recursively Constructed Sequences We will refer to the general inductive proof of the Bezout's Identity, turning a inductive proof into a constructive algorithm. Note that detecting the end of the loop involves one more iteration of the algorithm (k) than as stated in inductive proof ($k-1$).

Via Sequences with a second order recursion rule.
We compute the β_j with a recursion rule going from 0 up to $k+1$ (reading from right to left)

$$0 = \beta_{k+1} \qquad d = \beta_k \qquad\qquad (q_j,\, \beta_{j+1}) = \beta_{j-1} \operatorname{divMod} \beta_j \qquad\qquad \beta_1 = b \qquad \beta_0 = a$$

The supplemental q_j are computed alongside the β_j for 1 up to k.
We can use the q_j to espress the recursion relation on the β_j in reverse time going from $k+1$ back to 0.
We will group the boundary conditions and lay this out from right to left and bottom to top

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} \qquad\qquad \beta_{j-1} = q_j\beta_j + \beta_{j+1} \qquad\qquad \begin{bmatrix} \beta_k \\ \beta_{k+1} \end{bmatrix} = \begin{bmatrix} d \\ 0 \end{bmatrix}$$

Using the q_j we compute the ν_j by going from $k+1$ back to 0, one sequence for each of two cases (*) and (**) as distinguished by their initial conditions

$$\begin{bmatrix} m \\ n \end{bmatrix} = \begin{bmatrix} \nu_0 \\ \nu_1 \end{bmatrix} \qquad\qquad \nu_{j-1} = q_j\nu_j + \nu_{j+1} \qquad\qquad \begin{bmatrix} \nu_k \\ \nu_{k+1} \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}^{**} \text{ or } \begin{bmatrix} 0 \\ 1 \end{bmatrix}^{*}$$

We distinguish cases only as strictly necessary. Note that $0 < \nu_j < {}^{\beta_j}/_{d}$ whenever $j < k$; the looser bounds $0 \leqslant \nu_k \leqslant {}^{\beta_k}/_{d}$ hold at the penultimate $j = k$ and the final sequence element $\nu_{k+1} = \pm 1$ is the only ν_j which may not be a natural number.
Note that the recurrence relation mirrors that of the β_j going backwards in time $\beta_{j-1} = q_j\beta_j + \beta_{j+1}$.

We define the sequence of determinants δ_j with indices from k down to 0

$$\delta_j = \begin{vmatrix} \nu_j & \beta_j \\ \nu_{j+1} & \beta_{j+1} \end{vmatrix} = \nu_j\beta_{j+1} - \nu_{j+1}\beta_j$$

For each of two cases (**) and (*) we get a sequence which starts

$$\delta_k = \begin{vmatrix} \nu_k & d \\ \nu_{k+1} & 0 \end{vmatrix} = -\nu_{k+1}d \qquad \delta_k^{**} = \begin{vmatrix} 1 & d \\ -1 & 0 \end{vmatrix} = +d \qquad \delta_k^{*} = \begin{vmatrix} 0 & d \\ 1 & 0 \end{vmatrix} = -d$$

Which then oscillates between $\pm d$

$$\delta_{j-1} = \begin{vmatrix} \nu_{j-1} & \beta_{j-1} \\ \nu_j & \beta_j \end{vmatrix} = \begin{vmatrix} q_j\nu_j + \nu_{j+1} & q_j\beta_j + \beta_{j+1} \\ \nu_j & \beta_j \end{vmatrix} = \cancel{q_j \begin{vmatrix} \nu_j & \beta_j \\ \nu_j & \beta_j \end{vmatrix}} + \begin{vmatrix} \nu_{j+1} & \beta_{j+1} \\ \nu_j & \beta_j \end{vmatrix} = - \begin{vmatrix} \nu_j & \beta_j \\ \nu_{j+1} & \beta_{j+1} \end{vmatrix} = -\delta_j$$

$$\therefore \delta_0 = (-1)^k \cdot \delta_k$$

And which terminate with expressions δ_0 in the m and n coefficients

$$\begin{vmatrix} m & a \\ n & b \end{vmatrix} = \delta_0 \qquad \begin{vmatrix} m^{**} & a \\ n^{**} & b \end{vmatrix} = \delta_0^{**} = (-1)^k \cdot d \qquad \begin{vmatrix} m^{*} & a \\ n^{*} & b \end{vmatrix} = \delta_0^{*} = -(-1)^k \cdot d$$

When k is even $\delta_0^{**} > 0 > \delta_0^{*}$ and when k is odd $\delta_0^{*} > 0 > \delta_0^{**}$. The negative case can be reshuffled so that when $mb - na = -d < 0$ we reverse the terms to get a natural number expression $an - bm = d > 0$

$$\text{When } k \text{ is even} \quad \begin{vmatrix} m^{**} & a \\ n^{**} & b \end{vmatrix} = d = \begin{vmatrix} a & m^{*} \\ b & n^{*} \end{vmatrix} \qquad\qquad \text{When } k \text{ is odd} \quad \begin{vmatrix} m^{*} & a \\ n^{*} & b \end{vmatrix} = d = \begin{vmatrix} a & m^{**} \\ b & n^{**} \end{vmatrix}$$

Whether k is odd or even is already known before computing the ν_j sequences.
If only the positive δ_0 solution is required then only one of the ν_j sequences need be computed.

If we need both cases, we can combine and interleave the two sequences swapping as we go. This interleaving mirrors the behaviour of the recursive algorithm from the inductive proof. We must do this in the recursive algorithm because we dont know when the recursion will end and which case to pursue. In the left code examples below we will interleave cases for comparison purposes and also for brevity. The other code on the right, which splits out the two cases, is longer than it need be in order to highlight the independence of the cases. □

An Iterative Implementation of the Two-Phase Algorithm in Python The calculations implied by the general-inductive proof are the the same as those specified by the recursive sequence definitions for β_j and q_j followed by all the ν_j^* and/or ν_j^{**}. The following deconstructs the purely functional code of the previous example into stateful looping and only saves as intermediate data the sequence of quotients necessary to link the two phases together. We prepend the $\gcd(0,0)$ test for consistency

$(a,b) \mapsto d, ({}^a/_d, {}^b/_d), (mm, nn), (m,n)$ *where* d *is* $\gcd(a,b) = \left|\begin{smallmatrix} mm & nn \\ a & b \end{smallmatrix}\right| = \left|\begin{smallmatrix} a & b \\ m & n \end{smallmatrix}\right|$

```
def euclidean_bezout_2(b_, b, /):
    if b_ == 0 and b == 0:
        return 0, (0, 0), (-1, 1), (1, -1)
```

```
    # left variant mimics the recursive code
    quotients = []

    while b != 0:
        (q, b), b_ = divmod(b_, b), b
        quotients.append(q)

    # interleave cases (**) and (*)
    (nn, nn_), (n, n_) = (1, -1), (0, 1)
    for q in reversed(quotients):
        (nn, nn_), (n, n_) =\
          (q*n + n_, n), (q*nn + nn_, nn)
```

```
    # right variant calculates cases independently
    quotients = []
    p = True
    while b != 0:
        (q, b), b_ = divmod(b_, b), b
        quotients.append(q)
        p = not p

    # only calculate one of the cases (**) or (*)
    def c(ss):
        u, u_ = (1, -1) if ss else (0, 1)
        for q in reversed(quotients):
            u, u_ = q*u + u_, u
        return u, u_

    (nn, nn_), (n, n_) = c(p), c(not p)
```

```
    return b_, (nn + n, nn_ + n_), (nn, nn_), (n, n_)
```

The code loop that generates the quotients list, in addition to the list itself, only uses the two state variables of the Euclidean Algorithm the natural number b and its older state b_. In the code on the left the for loop which generates the Bezout coefficients uses the natural numbers n and nn and its older state n_ and nn_. Although it performs exactly the same computations while consuming fewer resources than the purely functional recursive code, there are still no bounds on how long the list of stored quotients will be. A following matrix analysis will show us how to limit the size of our stored quotients to that of a single 2×2 square matrix. But even then our algorithm lacks elegance — there are better alternatives.

In the code on the right the cases are calculated independently for expository purposes. This is a variant of the much the same code which takes advantage of knowing which sequence of ν_j to compute, then (for consistency with other code examples) throws that advantage away by returning both cases anyway. The added parity bit p is True when the number of quotients is even and False when the number is odd.

The ratio in least terms, the nnn in our recursive code example, are calculated at the return. ◇

Further Elaboration on the Recursion of the Two-Phase Algorithm Some Matrix Algebra should elaborate steps of the algorithm.

The recursion relation for β_j via the q_j (reading right to left)
(applying a square 2×2 matrix as a function to a column vector is simply matrix multiplication)

$$\begin{bmatrix} d \\ 0 \end{bmatrix} = \begin{bmatrix} \beta_k \\ \beta_{k+1} \end{bmatrix} \qquad \begin{bmatrix} \beta_j \\ \beta_{j+1} \end{bmatrix} = \begin{bmatrix} \beta_j \\ \beta_{j-1} - q_j\beta_j \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_j \end{smallmatrix}\right]} \begin{bmatrix} \beta_{j-1} \\ \beta_j \end{bmatrix} \qquad \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}$$

Unrolled this becomes a chain of matrix multiplications

$$\begin{bmatrix} d \\ 0 \end{bmatrix} = \begin{bmatrix} \beta_k \\ \beta_{k+1} \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_k \end{smallmatrix}\right]} \begin{bmatrix} \beta_{k-1} \\ \beta_k \end{bmatrix} \cdots \begin{bmatrix} \beta_j \\ \beta_{j+1} \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_j \end{smallmatrix}\right]} \begin{bmatrix} \beta_{j-1} \\ \beta_j \end{bmatrix} \cdots \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_1 \end{smallmatrix}\right]} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}$$

Written within the matrix notation proper

$$\begin{bmatrix} \beta_j \\ \beta_{j+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -q_{j-1} \end{bmatrix} \cdots \begin{bmatrix} 0 & 1 \\ 1 & -q_2 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -q_1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

The recursion relation in reverse time for the β_j and rhe recursion relation for the coefficients ν_j in cases $(**)$ and $(*)$ and in the switched cases $(+)$ and $(-)$ depending on k (and still reading right to left)

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} \qquad \begin{bmatrix} \beta_{j-1} \\ \beta_j \end{bmatrix} = \begin{bmatrix} q_j\beta_j + \beta_{j+1} \\ \beta_j \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} q_j & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} \beta_j \\ \beta_{j+1} \end{bmatrix} \qquad \begin{bmatrix} \beta_k \\ \beta_{k+1} \end{bmatrix} = \begin{bmatrix} d \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} m \\ n \end{bmatrix} = \begin{bmatrix} \nu_0 \\ \nu_1 \end{bmatrix} \qquad \begin{bmatrix} \nu_{j-1} \\ \nu_j \end{bmatrix} = \begin{bmatrix} q_j\nu_j + \nu_{j+1} \\ \nu_j \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} q_j & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} \nu_j \\ \nu_{j+1} \end{bmatrix} \qquad \begin{bmatrix} \nu_k \\ \nu_{k+1} \end{bmatrix} = \begin{cases} \left[\begin{smallmatrix} 1 \\ -1 \end{smallmatrix}\right] & (**) \Leftarrow \begin{cases} (+) & k \text{ even} \\ (-) & k \text{ odd} \end{cases} \\ \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right] & (*) \Leftarrow \begin{cases} (-) & k \text{ even} \\ (+) & k \text{ odd} \end{cases} \end{cases}$$

As a sequential application of the iterative steps $\left[\begin{smallmatrix} q_j & 1 \\ 1 & 0 \end{smallmatrix}\right]$ back from $\left[\begin{smallmatrix} d \\ 0 \end{smallmatrix}\right]$ or $\left[\begin{smallmatrix} \nu_k \\ \nu_{k+1} \end{smallmatrix}\right]$

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} q_1 & 1 \\ 1 & 0 \end{smallmatrix}\right]} \cdots \xleftarrow{\left[\begin{smallmatrix} q_{k-1} & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} \beta_{k-1} \\ \beta_k \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} q_k & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} \beta_k \\ \beta_{k+1} \end{bmatrix} = \begin{bmatrix} d \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} m \\ n \end{bmatrix} = \begin{bmatrix} \nu_0 \\ \nu_1 \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} q_1 & 1 \\ 1 & 0 \end{smallmatrix}\right]} \cdots \xleftarrow{\left[\begin{smallmatrix} q_{k-1} & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} \nu_{k-1} \\ \nu_k \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} q_k & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} \nu_k \\ \nu_{k+1} \end{bmatrix}$$

Applying the total $\left[\begin{smallmatrix} q_1 & 1 \\ 1 & 0 \end{smallmatrix}\right] \left[\begin{smallmatrix} q_2 & 1 \\ 1 & 0 \end{smallmatrix}\right] \cdots \left[\begin{smallmatrix} q_{k-1} & 1 \\ 1 & 0 \end{smallmatrix}\right] \left[\begin{smallmatrix} q_k & 1 \\ 1 & 0 \end{smallmatrix}\right]$ transformation to the specific $\left[\begin{smallmatrix} \nu_k \\ \nu_{k+1} \end{smallmatrix}\right]$ seeds

$$\begin{bmatrix} m^{**} \\ n^{**} \end{bmatrix} \leftarrow \begin{bmatrix} 1 \\ -1 \end{bmatrix} \qquad \begin{bmatrix} m^{*} \\ n^{*} \end{bmatrix} \leftarrow \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad \begin{bmatrix} m^{+} \\ n^{+} \end{bmatrix} \leftarrow \begin{cases} \left[\begin{smallmatrix} 1 \\ -1 \end{smallmatrix}\right] k \text{ even} \\ \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right] k \text{ odd} \end{cases} \qquad \begin{bmatrix} m^{-} \\ n^{-} \end{bmatrix} \leftarrow \begin{cases} \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right] k \text{ even} \\ \left[\begin{smallmatrix} 1 \\ -1 \end{smallmatrix}\right] k \text{ odd} \end{cases}$$

Concatenating the transformation of the two cases (*) and (**) to get a square 2×2 matrix

$$\begin{bmatrix} m^{**} & m^{*} \\ n^{**} & n^{*} \end{bmatrix} = \begin{bmatrix} q_1 & 1 \\ 1 & 0 \end{bmatrix} \cdots \begin{bmatrix} q_k & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \qquad \begin{bmatrix} m^{+} & m^{-} \\ n^{+} & n^{-} \end{bmatrix} = \begin{cases} \left[\begin{smallmatrix} m^{**} & m^{*} \\ n^{**} & n^{*} \end{smallmatrix}\right] \text{ when } k \text{ is even} \\ \left[\begin{smallmatrix} m^{*} & m^{**} \\ n^{*} & n^{**} \end{smallmatrix}\right] \text{ when } k \text{ is odd} \end{cases}$$

Multiplying on the right by the $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$ matrix k times swaps columns (every second swap undoes the first)

$$\begin{aligned} \begin{bmatrix} m^{+} & m^{-} \\ n^{+} & n^{-} \end{bmatrix} &= \begin{bmatrix} m^{**} & m^{*} \\ n^{**} & n^{*} \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^k \\ &= \begin{bmatrix} q_1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} q_2 & 1 \\ 1 & 0 \end{bmatrix} \cdots \begin{bmatrix} q_{k-1} & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} q_k & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^k \end{aligned}$$

Viewing the $\left[\begin{smallmatrix} m^+ & m^- \\ n^+ & n^- \end{smallmatrix}\right]$ matrix as a transformation in toto

$$\begin{bmatrix} m^+ \\ n^+ \end{bmatrix} = \left\{\begin{matrix} \left[\begin{smallmatrix} m^{**} \\ n^{**} \end{smallmatrix}\right] \\ \left[\begin{smallmatrix} m^{*} \\ n^{*} \end{smallmatrix}\right] \end{matrix}\right\} \xleftarrow{\left[\begin{smallmatrix} q_1 & 1 \\ 1 & 0 \end{smallmatrix}\right] \cdots \left[\begin{smallmatrix} q_k & 1 \\ 1 & 0 \end{smallmatrix}\right]} \left\{\begin{matrix} \left[\begin{smallmatrix} 1 \\ -1 \end{smallmatrix}\right] \\ \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right] \end{matrix}\right\} \xleftarrow{\left[\begin{smallmatrix} 1 & 0 \\ -1 & 1 \end{smallmatrix}\right]} \left\{\begin{matrix} \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right] k \text{ even} \\ \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right] k \text{ odd} \end{matrix}\right\} \xleftarrow{\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]^k} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} m^- \\ n^- \end{bmatrix} = \left\{\begin{matrix} \left[\begin{smallmatrix} m^{*} \\ n^{*} \end{smallmatrix}\right] \\ \left[\begin{smallmatrix} m^{**} \\ n^{**} \end{smallmatrix}\right] \end{matrix}\right\} \xleftarrow{\left[\begin{smallmatrix} q_1 & 1 \\ 1 & 0 \end{smallmatrix}\right] \cdots \left[\begin{smallmatrix} q_k & 1 \\ 1 & 0 \end{smallmatrix}\right]} \left\{\begin{matrix} \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right] \\ \left[\begin{smallmatrix} 1 \\ -1 \end{smallmatrix}\right] \end{matrix}\right\} \xleftarrow{\left[\begin{smallmatrix} 1 & 0 \\ -1 & 1 \end{smallmatrix}\right]} \left\{\begin{matrix} \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right] k \text{ even} \\ \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right] k \text{ odd} \end{matrix}\right\} \xleftarrow{\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]^k} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} a \\ b \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} q_1 & 1 \\ 1 & 0 \end{smallmatrix}\right]\left[\begin{smallmatrix} q_2 & 1 \\ 1 & 0 \end{smallmatrix}\right] \cdots \left[\begin{smallmatrix} q_{k-1} & 1 \\ 1 & 0 \end{smallmatrix}\right]\left[\begin{smallmatrix} q_k & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} d \\ 0 \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} 1 & 0 \\ -1 & 1 \end{smallmatrix}\right]} \begin{bmatrix} d \\ d \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]^k} \begin{bmatrix} d \\ d \end{bmatrix}$$

Computing the determinant of $\left[\begin{smallmatrix} m^+ & m^- \\ n^+ & n^- \end{smallmatrix}\right]$

$$\begin{aligned}
\begin{vmatrix} m^+ & m^- \\ n^+ & n^- \end{vmatrix} &= \det\left(\begin{bmatrix} q_1 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} q_2 & 1 \\ 1 & 0 \end{bmatrix} \cdots \begin{bmatrix} q_{k-1} & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} q_k & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^k\right) \\
&= \underbrace{\underbrace{\begin{vmatrix} q_1 & 1 \\ 1 & 0 \end{vmatrix}}_{-1}\underbrace{\begin{vmatrix} q_2 & 1 \\ 1 & 0 \end{vmatrix}}_{-1} \cdots \underbrace{\begin{vmatrix} q_{k-1} & 1 \\ 1 & 0 \end{vmatrix}}_{-1}\underbrace{\begin{vmatrix} q_k & 1 \\ 1 & 0 \end{vmatrix}}_{-1}}_{\cancel{(-1)^k}}\underbrace{\begin{vmatrix} 1 & 0 \\ -1 & 1 \end{vmatrix}}_{1}\underbrace{\det\left(\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^k\right)}_{\cancel{(-1)^k}} \\
&= 1
\end{aligned}$$

For using Cramer's Rule

$$\begin{bmatrix} m^+ & m^- \\ n^+ & n^- \end{bmatrix}\begin{bmatrix} d \\ d \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \quad \text{and} \quad \begin{vmatrix} m^+ & m^- \\ n^+ & n^- \end{vmatrix} = 1 \quad \Longrightarrow \quad \begin{vmatrix} m^+ & a \\ n^+ & b \end{vmatrix} = d = \begin{vmatrix} a & m^- \\ b & n^- \end{vmatrix}$$

Or using the determinant and adjoint matrix of $\left[\begin{smallmatrix} m^+ & m^- \\ n^+ & n^- \end{smallmatrix}\right]$ for the 2×2 matrix inversion formula

$$\begin{bmatrix} m^+ & m^- \\ n^+ & n^- \end{bmatrix}^{-1} = \begin{bmatrix} +n^- & -m^- \\ -n^+ & +m^+ \end{bmatrix} \times \cancel{\frac{1}{\left|\begin{smallmatrix} m^+ & m^- \\ n^+ & n^- \end{smallmatrix}\right|}}$$

$$\therefore \begin{bmatrix} d \\ d \end{bmatrix} = \begin{bmatrix} m^+ & m^- \\ n^+ & n^- \end{bmatrix}^{-1}\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} +n^- & -m^- \\ -n^+ & +m^+ \end{bmatrix}\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} n^-a - m^-b \\ m^+b - n^+a \end{bmatrix}$$

Using the 2×2 matrix inversion formula or using Cramer's Rule allows us to bypass the δ_j sequence and demonstrate the efficacy of the computed Bezout's Identity coefficients in a single step. □

A Bounded Iterative Implementation of the Two-Phase Algorithm in Python While it would be easy enough to change our code to save a quotients matrix rather than a list of quotients, this code would highlight a mismatch between our recursively defined sequences and our implementation. Instead, we will go back to basics for greater clarity. ◇

Corollary: A One-Phase Algorithm via Recursively Constructed Sequences It is possible to derive the m and n coefficients without calculating back from the ν_{k+1} and ν_k.

Via Sequences with a second order recursion rule.
We define two sequences of natural numbers v_j and w_j that progress forward alongside the β_j.
For β_j, v_j and w_j going from 0 up to $k+1$ (reading from right to left and inline for a compact presentation)

$$\begin{array}{rcc} (q_j, \beta_{j+1}) = \beta_{j-1} \text{ divMod } \beta_j & \beta_1 = b & \beta_0 = a \\ v_{j+1} = q_j v_j + v_{j-1} & v_1 = 1 & v_0 = 0 \\ w_{j+1} = q_j w_j + w_{j-1} & w_1 = 0 & w_0 = 1 \end{array}$$

The β_j terminate with $\beta_k = d$ and $\beta_{k+1} = 0$. Note that the β_j are strictly decreasing and the v_j and w_j are strictly increasing for $j > 0$ (or $j > 1$ if the first iteration is a trivial swap of a and b when $a < b$).

The state of β forms a column vector $\left[\begin{smallmatrix} \beta_j \\ \beta_{j+1} \end{smallmatrix}\right]$ starting $\left[\begin{smallmatrix} \beta_0 \\ \beta_1 \end{smallmatrix}\right] = \left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$ and terminating with $\left[\begin{smallmatrix} d \\ 0 \end{smallmatrix}\right] = \left[\begin{smallmatrix} \beta_k \\ \beta_{k+1} \end{smallmatrix}\right]$.
This β state is advanced by multipying the column vector by the $\left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_j \end{smallmatrix}\right]$ matrix on the left or is regressed by multipying by the inverse $\left[\begin{smallmatrix} q_j & 1 \\ 1 & 0 \end{smallmatrix}\right]$ matrix on the left.

The state of v forms a row vector $\begin{bmatrix} v_{j+1} & v_j \end{bmatrix}$. Likewise for w we have a row vector $\begin{bmatrix} w_{j+1} & w_j \end{bmatrix}$.
Multiplying by a $\left[\begin{smallmatrix} q_j & 1 \\ 1 & 0 \end{smallmatrix}\right]$ matrix on the right advances that state
(function application of a 2×2 matrix on a row vector is typically written left-to-right)

$$\begin{bmatrix} v_j & v_{j-1} \end{bmatrix} \xrightarrow{\left[\begin{smallmatrix} q_j & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} v_{j+1} & v_j \end{bmatrix} \qquad \begin{bmatrix} v_1 & v_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} w_j & w_{j-1} \end{bmatrix} \xrightarrow{\left[\begin{smallmatrix} q_j & 1 \\ 1 & 0 \end{smallmatrix}\right]} \begin{bmatrix} w_{j+1} & w_j \end{bmatrix} \qquad \begin{bmatrix} w_1 & w_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

Putting one atop the other forms a square matrix and its defining recursion relation (reading right to left)

$$\begin{bmatrix} v_{j+1} & v_j \\ w_{j+1} & w_j \end{bmatrix} = \begin{bmatrix} v_j & v_{j-1} \\ w_j & w_{j-1} \end{bmatrix} \begin{bmatrix} q_j & 1 \\ 1 & 0 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} v_1 & v_0 \\ w_1 & w_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

As a repeated product

$$\begin{bmatrix} v_{j+1} & v_j \\ w_{j+1} & w_j \end{bmatrix} = \begin{bmatrix} q_1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} q_2 & 1 \\ 1 & 0 \end{bmatrix} \cdots \begin{bmatrix} q_{j-1} & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} q_j & 1 \\ 1 & 0 \end{bmatrix}$$

For the ongoing relationship between the v, w and β sequences

$$\begin{bmatrix} a \\ b \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} v_{j+1} & v_j \\ w_{j+1} & w_j \end{smallmatrix}\right]} \begin{bmatrix} \beta_j \\ \beta_{j+1} \end{bmatrix} \quad \text{or conversely} \quad \begin{bmatrix} \beta_j \\ \beta_{j+1} \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} v_{j+1} & v_j \\ w_{j+1} & w_j \end{smallmatrix}\right]^{-1}} \begin{bmatrix} a \\ b \end{bmatrix}$$

The matrices associated with the advancing state of the v and w all have a determinant of minus one

$$\begin{vmatrix} v_{j+1} & v_j \\ w_{j+1} & w_j \end{vmatrix} = (-1)(-1)\cdots(-1)(-1) = (-1)^j$$

Making the v, w state matrix invertible within the integers using the 2×2 matrix inversion formula

$$\begin{bmatrix} v_{j+1} & v_j \\ w_{j+1} & w_j \end{bmatrix}^{-1} = \begin{bmatrix} +w_j & -v_j \\ -w_{j+1} & +v_{j+1} \end{bmatrix} (-1)^j = \begin{cases} \left[\begin{smallmatrix} +w_j & -v_j \\ -w_{j+1} & +v_{j+1} \end{smallmatrix}\right] & j \text{ even} \\ \left[\begin{smallmatrix} -w_j & +v_j \\ +w_{j+1} & -v_{j+1} \end{smallmatrix}\right] & j \text{ odd} \end{cases}$$

$$\begin{bmatrix} \beta_j \\ \beta_{j+1} \end{bmatrix} = \begin{bmatrix} +w_j & -v_j \\ -w_{j+1} & +v_{j+1} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} (-1)^j = \begin{cases} \left[\begin{smallmatrix} +w_j & -v_j \\ -w_{j+1} & +v_{j+1} \end{smallmatrix}\right] \left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right] & j \text{ even} \\ \left[\begin{smallmatrix} -w_j & +v_j \\ +w_{j+1} & -v_{j+1} \end{smallmatrix}\right] \left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right] & j \text{ odd} \end{cases}$$

When the underlying v and w are non-zero (i.e. after two nontrivial iterations of the algorithm) the inverse state matrix always contains exactly two positive and two negative elements. For this algorithm, despite its seeming utility, we never need the inverse state matrix or its action on $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$.

And for the final relationship between a, b and d

$$\begin{bmatrix} +w_k & -v_k \\ -w_{k+1} & +v_{k+1} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} (-1)^k = \begin{bmatrix} d \\ 0 \end{bmatrix} \quad \overset{\text{suggestive but unneeded}}{\Longrightarrow} \quad \begin{aligned} \big(w_k a - v_k b\big)(-1)^k &= d \\ w_{k+1} a &= v_{k+1} b \end{aligned}$$

$$\begin{bmatrix} v_{k+1} & v_k \\ w_{k+1} & w_k \end{bmatrix} \begin{bmatrix} d \\ 0 \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \quad \overset{\text{seemingly dull but actually useful}}{\Longrightarrow} \quad \begin{aligned} v_{k+1} d &= a \\ w_{k+1} d &= b \end{aligned}$$

Note that the inverse of the v, w state matrix only reveals one of the Bezout solutions. Instead we define an m, n matrix by tacking onto the end of the final v, w state to generate both Bezout's Identity solutions in a elegent manner using Cramer's Rule

$$\begin{bmatrix} m^+ & m^- \\ n^+ & n^- \end{bmatrix} = \begin{bmatrix} v_{k+1} & v_k \\ w_{k+1} & w_k \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^k = \begin{bmatrix} v_{k+1}{-}v_k & v_k \\ w_{k+1}{-}w_k & w_k \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}^k = \begin{cases} \left[\begin{smallmatrix} v_{k+1}-v_k & v_k \\ w_{k+1}-w_k & w_k \end{smallmatrix}\right] & k \text{ even} \\ \left[\begin{smallmatrix} v_k & v_{k+1}-v_k \\ w_k & w_{k+1}-w_k \end{smallmatrix}\right] & k \text{ odd} \end{cases}$$

$$\begin{vmatrix} m^+ & m^- \\ n^+ & n^- \end{vmatrix} = (-1)^k \times 1 \times (-1)^k = 1$$

$$\begin{bmatrix} m^+ & m^- \\ n^+ & n^- \end{bmatrix} \begin{bmatrix} d \\ d \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} v_{k+1} & v_k \\ w_{k+1} & w_k \end{smallmatrix}\right]} \begin{bmatrix} d \\ 0 \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} 1 & 0 \\ -1 & 1 \end{smallmatrix}\right]} \begin{bmatrix} d \\ d \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]^k} \begin{bmatrix} d \\ d \end{bmatrix} \overset{\text{Cramer's Rule}}{\Longrightarrow} \begin{vmatrix} m^+ & a \\ n^+ & b \end{vmatrix} = d = \begin{vmatrix} a & m^- \\ b & n^- \end{vmatrix}$$

That the β_j sequences terminate with the greatest common divisor d and 0 is clear. This together with the increasing nature of the v_j and w_j sequences together with the boundary values of the $v_{k+1} = {}^a\!/\!_d$ and $w_{k+1} = {}^b\!/\!_d$ is sufficient proof for Bézout's Identity — the inductive proof isn't really needed. But the inductive proof does justify where this somewhat magical algorithm comes from. □

An Iterative Implementation of the One-Phase Algorithm in Python This code takes the same arguments and returns exactly the same values as does the other python code, despite coming at those values by different intermediate computations

$(a, b) \mapsto d, ({}^a\!/\!_d, {}^b\!/\!_d), (mm, nn), (m, n)$ *where* d *is* $\gcd(a, b) = \left|\begin{smallmatrix} mm & nn \\ a & b \end{smallmatrix}\right| = \left|\begin{smallmatrix} a & b \\ m & n \end{smallmatrix}\right|$

```
def euclidean_bezout_3(b_, b, /):
    if b_ == 0 and b == 0:
        return 0, (0, 0), (-1, 1), (1, -1)

    v, v_ = 1, 0
    w, w_ = 0, 1
    p = True
    while b != 0:
        (q, b), b_ = divmod(b_, b), b
        v, v_ = q*v + v_, v
        w, w_ = q*w + w_, w
        p = not p
    if p:
        return b_, (v, w), (v - v_, w - w_), (v_, w_)
    else:
        return b_, (v, w), (v_, w_), (v - v_, w - w_)
```

This code is very parsimonious, very well-behaved requiring only a small and fixed amount of state, being the natural number `b`, its older state `b_`, the natural number `v`, its older state `v_`, the natural number `w`, its older state `w_` and the parity bit `p`. ◇

Corollary: Another One-Phase Algorithm via a Recursively Constructed 2×3 Matrix We define two sequences of integers x_j and y_j that progress forward alongside and in the same manner as the natural valued β_j. In a straightforward and ongoing way we maintain the identity $\beta_j = x_j a + y_j b$. For β_j, x_j and y_j going from 0 up to $k+1$

$$\begin{aligned} (q_j, \beta_{j+1}) &= \beta_{j-1} \operatorname{divMod} \beta_j & \beta_1 &= b & \beta_0 &= a \\ x_{j+1} &= x_{j-1} - q_j x_j & x_1 &= 1 & x_0 &= 0 \\ y_{j+1} &= y_{j-1} - q_j y_j & y_1 &= 0 & y_0 &= 1 \end{aligned}$$

Which can be thought of as advancing the state of a 2×3 matrix $\mathbf{B}_j$ of integer (not just natural) values

$$\mathbf{B}_j = \left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_j \end{smallmatrix}\right] \mathbf{B}_{j-1} \longleftarrow \mathbf{B}_{j-1} \qquad \mathbf{B}_0 = \begin{bmatrix} \beta_0 & x_0 & y_0 \\ \beta_1 & x_1 & y_1 \end{bmatrix} = \left[\begin{array}{c|cc} a & 1 & 0 \\ b & 0 & 1 \end{array}\right]$$

$$\mathbf{B}_k = \left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_k \end{smallmatrix}\right] \left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_{k-1} \end{smallmatrix}\right] \cdots \left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_2 \end{smallmatrix}\right] \left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_1 \end{smallmatrix}\right] \mathbf{B}_0$$

Component-wise the iterative mapping via $\left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_j \end{smallmatrix}\right]$ only generates one new row per generation

$$\mathbf{B}_j = \begin{bmatrix} \beta_j & x_j & y_j \\ \beta_{j+1} & x_{j+1} & y_{j+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix} \begin{bmatrix} \beta_{j-1} & x_{j-1} & y_{j-1} \\ \beta_j & x_j & y_j \end{bmatrix} \longleftarrow \begin{bmatrix} \beta_{j-1} & x_{j-1} & y_{j-1} \\ \beta_j & x_j & y_j \end{bmatrix} = \mathbf{B}_{j-1}$$

As a repeated product component-wise

$$\mathbf{B}_j = \left[\begin{array}{c|cc} \beta_j & x_j & y_j \\ \beta_{j+1} & x_{j+1} & y_{j+1} \end{array}\right] = \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix} \cdots \begin{bmatrix} 0 & 1 \\ 1 & -q_1 \end{bmatrix} \left[\begin{array}{c|cc} a & 1 & 0 \\ b & 0 & 1 \end{array}\right]$$

$$\begin{bmatrix} x_j & y_j \\ x_{j+1} & y_{j+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix} \cdots \begin{bmatrix} 0 & 1 \\ 1 & -q_1 \end{bmatrix}$$

The submatrix of x, y has a predictable pattern of signedness depending on whether k is odd or even. Combined with the action of the matrices $\left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_j \end{smallmatrix}\right]$ this results in the x_j and y_j bouncing in sign but strictly increasing in magnitude for $j > 0$ (or $j > 1$ if the first iteration is a trivial swap of a and b when $a < b$). This 2×3 matrix $\mathbf{B}_j$ effectively represents the ongoing relationship

$$\mathbf{B}_j = \begin{bmatrix} \beta_j & x_j & y_j \\ \beta_{j+1} & x_{j+1} & y_{j+1} \end{bmatrix} \iff \begin{bmatrix} \beta_j \\ \beta_{j+1} \end{bmatrix} = \begin{bmatrix} x_j & y_j \\ x_{j+1} & y_{j+1} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

So that, with the final state of the 2×3 matrix $\mathbf{B}_k$

$$\mathbf{B}_k = \left[\begin{array}{c|cc} d & x_k & y_k \\ 0 & x_{k+1} & y_{k+1} \end{array}\right] \implies \begin{bmatrix} d \\ 0 \end{bmatrix} = \begin{bmatrix} x_k & y_k \\ x_{k+1} & y_{k+1} \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x_k a + y_k b \\ x_{k+1} a + y_{k+1} b \end{bmatrix}$$

The β_j sequences terminate with the gcd d and 0. With the increasing magnitude of the x_j and y_j sequences together with the boundary values on the x_{k+1} and y_{k+1} would prove Bézout's Identity; but these boundary values, $|x_{k+1}| = {}^{b}/_{d}$ and $|y_{k+1}| = {}^{a}/_{d}$, are inferred via the inverse to the x, y state matrix which this particular construction provides no justification for whatsoever. This would be truly magical to insert here in the development of the algorithm.

We will append to this terminal state $\mathbf{B}_k$ to generate both set of solutions in a matrix $\mathbf{B}_\pm$

$$\begin{bmatrix} d \\ d \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]^k} \begin{bmatrix} d \\ d \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right]} \begin{bmatrix} d \\ 0 \end{bmatrix} \xleftarrow{\left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_k \end{smallmatrix}\right] \left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_{k-1} \end{smallmatrix}\right] \cdots \left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_2 \end{smallmatrix}\right] \left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_1 \end{smallmatrix}\right]} \begin{bmatrix} a \\ b \end{bmatrix}$$

$$\mathbf{B}_\pm = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]^k \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right] \mathbf{B}_k = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]^k \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right] \left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_k \end{smallmatrix}\right] \left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_{k-1} \end{smallmatrix}\right] \cdots \left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_2 \end{smallmatrix}\right] \left[\begin{smallmatrix} 0 & 1 \\ 1 & -q_1 \end{smallmatrix}\right] \left[\begin{smallmatrix} a & 1 & 0 \\ b & 0 & 1 \end{smallmatrix}\right]$$

This $\mathbf{B}_\pm$ conforms to the following pattern of signedness (where $+$ means non-negative) $\begin{bmatrix} + & + & - \\ + & - & + \end{bmatrix}$

$$\begin{aligned}
\mathbf{B}_\pm = \left[\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right]^k \left[\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right] \mathbf{B}_k &= \begin{bmatrix}0&1\\1&0\end{bmatrix}^k \begin{bmatrix}1&0\\1&1\end{bmatrix} \begin{bmatrix}d & x_k & y_k \\ 0 & x_{k+1} & y_{k+1}\end{bmatrix} \\
&= \begin{bmatrix}0&1\\1&0\end{bmatrix}^k \begin{bmatrix}d & x_k & y_k \\ d & x_{k+1}+x_k & y_{k+1}+y_k\end{bmatrix} \\
&= \begin{cases} \left[\begin{smallmatrix}d & x_k & y_k \\ d & x_{k+1}+x_k & y_{k+1}+y_k\end{smallmatrix}\right] & \text{when } k \text{ is even} \\ \left[\begin{smallmatrix}d & x_{k+1}+x_k & y_{k+1}+y_k \\ d & x_k & y_k\end{smallmatrix}\right] & \text{when } k \text{ is odd} \end{cases}
\end{aligned}$$

This algorithm is quite strange. That the identity holds is clear, but the boundedness and signedness of the coefficients is quite mysterious. The appearance of the swap matrix is understandable but the $\left[\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right]$ matrix appears in a *deus ex machina*. If we hadn't already seen the the submatrix of x, y as the inverse the natural-valued and easily understood v, w matrix we would be hard pressed to justify the assertions we have made. □

A Matrix Implementation of the One-Phase Algorithm in Python This code is tricky because of the different assumptions made with respect to its returned coefficients. In the development of the algorithm the coefficients are required to be integers x and y such that $xa + yb = d$. In order to be consistent with our previous code, which expresses the Bezout Identity via a determinant, we will need to be very careful with respect to signedness and the order of coefficients. Had we developed this code first the return statement would be much simpler here and a little more complex in the other code examples

```
# $(a,b) \mapsto d, (a/d, b/d), (mm, nn), (m, n)$ where $d$ is $\gcd(a,b) = \left|\begin{smallmatrix} mm & nn \\ a & b\end{smallmatrix}\right| = \left|\begin{smallmatrix} a & b \\ m & n\end{smallmatrix}\right|$
def euclidean_bezout_4(a, b, /):
    if a == 0 and b == 0:
        return 0, (0, 0), (-1, 1), (1, -1)

    from numpy import array
    B, B_ = array((b, 0, 1)), array((a, 1, 0))
    p = True
    while B[0] != 0:
        q = B_[0] // B[0]
        B, B_ = B_ - q*B, B
        p = not p
    BB = B + B_
    if p:
        return B_[0], (B[2], -B[1]), (BB[2], -BB[1]), (-B_[2], B_[1])
    else:
        return B_[0], (-B[2], B[1]), (B_[2], -B_[1]), (-BB[2], BB[1])
```

The state is fully represented by the matrix row B, its previous state B_ and the parity bit p. This code may be more parsimonious than the previous example as numpy arrays are stored very compactly. ◇

Wrap-Up

We have seen four subtly different proofs of Bezout's Identity based around slightly different algorithms to compute the same final results. Given the recursive sequence definitions of the β_j and ν_j the behaviour of the δ_j proves Bezout's Identity; Likewise, the behaviour of the 2×2 q_j matrices when inverted proves the same.

The first proof of Bezout's Identity provided by a *one-phase iterative* algorithm (that computing the v, w state matrix alongside the β_j) by itself has very little justification and is quite *magical*. But the alternate one-phase algorithm which progressively strings together integer-valued q_j matrices (which are inverse to the the natural valued q_j we demonstrated earlier) to push forward the 2×3 state matrix $\mathbf{B}_j$ maintains a quite natural identity effortlessly proving an unbounded integer version of Bezout's Identity. But this creates a sequence of 2×2 state x, y submatrices having a alternating pattern of two positive and two negative elements; directly showing that the elements of this state matrix satisfy the upper bounds on the magnitude of the Bezout's Identity coefficients requires tracking the signedness of the matrix components to show the magnitudes are increasing and determining their ultimate bounding values — the 2×2 matrix inverse formula does make this clear, but only in the context of the first *one-phase* algorithm.

This final algorithm isn't so much magical as it is incomprehensible. It works. That the gcd results from a linear combination of the arguments is obvious at a glance. And anyone following it along step-by-step will be convinced that bounds on the magnitude of the Bezout's Identity coefficients will hold; but actually proving without context it isn't at all obvious. It's quite possible, but utterly mysterious. That this mystery algorithm is the most commonly presented is unfortunate. This final algorithm fails to treat Bezout's Identity as operating on and within the natural numbers, where bounds on the magnitude of the coefficients arise organically.

Our presentation is exhaustive enough than there should be no mysteries surrounding the effectiveness of the algorithms. The mathematics made available by Bezout's Identity is another matter entirely.