Modern ECC signatures

2011 Bernstein–Duif–Lange–Schwabe–Yang:
Ed25519 signature scheme = EdDSA using conservative
Curve25519 elliptic curve.
[https://ed25519.cr.yp.to](https://ed25519.cr.yp.to)

32-byte public keys,
64-byte signatures,
$\approx 2^{125.8}$ security level.

Deployed in SSH, Signal,
many more applications:

Many papers have explored
Curve25519/Ed25519 speed.
e.g. 2015 Chou software:
on Intel Sandy Bridge (2011),
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

Compare to, e.g., 2000 Brown–Hankerson–López–Menezes:
on Intel Pentium II (1997),
1920000 cycles for ECDH
using NIST P-256 curve.
Modern ECC signatures

2011 Bernstein–Duif–Lange–Schwabe–Yang:
Ed25519 signature scheme =
EdDSA using conservative
Curve25519 elliptic curve.

https://ed25519.cr.yp.to

32-byte public keys,
64-byte signatures,
≈ 2^{125} security level.

Deployed in SSH, Signal,
many more applications:
https://ianix.com/pub
/ed25519-deployment.html

Many papers have explored
Curve25519/Ed25519 speed.
e.g. 2015 Chou software:
on Intel Sandy Bridge (2011),
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

Compare to, e.g., 2000 Brown–Hankerson–López–Menezes:
on Intel Pentium II (1997),
1920000 cycles for ECDH
using NIST P-256 curve.

Does \( A_C \) : cycles for alg \( A \) on CPU \( C \) .
Does \( A_C < B_D \) prove that \( A \) is better than \( B \) ?
Many papers have explored Curve25519/Ed25519 speed.  

E.g. 2015 Chou software:  
on Intel Sandy Bridge (2011),  
57164 cycles for keygen,  
63526 cycles for signature,  
205741 cycles for verification,  
159128 cycles for ECDH.  

Compare to, e.g., 2000 Brown–Hankerson–López–Menezes:  
on Intel Pentium II (1997),  
1920000 cycles for ECDH  
using NIST P-256 curve.
Many papers have explored Curve25519/Ed25519 speed. e.g. 2015 Chou software: on Intel Sandy Bridge (2011), 57164 cycles for keygen, 63526 cycles for signature, 205741 cycles for verification, 159128 cycles for ECDH.


Does $A_C < B_D$ prove that $A$ is better than $B$?
Many papers have explored Curve25519/Ed25519 speed.

e.g. 2015 Chou software:
on Intel Sandy Bridge (2011),
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

Compare to, e.g., 2000 Brown–Hankerson–López–Menezes:
on Intel Pentium II (1997),
1920000 cycles for ECDH
using NIST P-256 curve.

$A_C$: cycles for alg $A$ on CPU $C$.
Does $A_C < B_D$ prove that $A$ is better than $B$?
Many papers have explored Curve25519/Ed25519 speed.
e.g. 2015 Chou software:
on Intel Sandy Bridge (2011),
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.
Compare to, e.g., 2000 Brown–Hankerson–López–Menezes:
on Intel Pentium II (1997),
1920000 cycles for ECDH
using NIST P-256 curve.

$A_C$: cycles for alg $A$ on CPU $C$.
Does $A_C < B_D$ prove that $A$ is better than $B$?
No! Beware change in CPU.
Maybe $A_C > B_C; A_D > B_D;$
$C$ does more work per cycle than $D$, thanks to CPU manufacturer.
Sometimes people measure cost
in seconds instead of cycles.
Then they benefit
from more work per cycle and
from more cycles per second.
Many papers have explored Curve25519/Ed25519 speed. e.g. 2015 Chou software: on Intel Sandy Bridge (2011), 57164 cycles for keygen, 63526 cycles for signature, 205741 cycles for verification, 159128 cycles for ECDH.


$A_C$: cycles for alg $A$ on CPU $C$. Does $A_C < B_D$ prove that $A$ is better than $B$?

No! Beware change in CPU.

Maybe $A_C > B_C; A_D > B_D$; $C$ does more work per cycle than $D$, thanks to CPU manufacturer.

Sometimes people measure cost in seconds instead of cycles. Then they benefit from more work per cycle and from more cycles per second.

Better comparisons (still raising many questions):

ECDH on Intel Pentium II/III (still not exactly the same): 1920000 cycles for NIST P-256, 832457 cycles for Curve25519.

ECDH on Sandy Bridge: 374000 cycles for NIST P-256 (from 2013 Gueron–Krasnov), 159128 cycles for Curve25519.

Verification on Sandy Bridge: 529000 cycles for ECDSA-P-256, 205741 cycles for Ed25519.
Many papers have explored Curve25519/Ed25519 speed. For instance, Chou's software on Intel Sandy Bridge (2011) achieved:
- 57,164 cycles for keygen,
- 63,526 cycles for signature,
- 205,741 cycles for verification,
- 159,128 cycles for ECDH.

- 1,920,000 cycles for ECDH using NIST P-256 curve.

$$A_C : \text{cycles for alg } A \text{ on CPU } C$$

Does $$A_C < B_D$$ prove that $$A$$ is better than $$B$$? No! Beware change in CPU.

Maybe $$A_C > B_C; A_D > B_D; C$$ does more work per cycle than $$D$$, thanks to CPU manufacturer.

Sometimes people measure cost in seconds instead of cycles. Then they benefit from more work per cycle and from more cycles per second.

Better comparisons (still raising many questions):

- ECDH on Intel Pentium II/III (still not exactly the same):
  - 1,920,000 cycles for NIST P-256,
  - 832,457 cycles for Curve25519.

- ECDH on Sandy Bridge:
  - 374,000 cycles for NIST P-256 (from 2013 Gueron–Krasnov),
  - 159,128 cycles for Curve25519.

- Verification on Sandy Bridge:
  - 529,000 cycles for ECDSA-P-256,
  - 2,057,41 cycles for Ed25519.
Many papers have explored Curve25519/Ed25519 speed. e.g. 2015 Chou software: on Intel Sandy Bridge (2011), 57164 cycles for keygen, 63526 cycles for signature, 205741 cycles for verification, 159128 cycles for ECDH.


\[ A_C : \text{cycles for alg } A \text{ on CPU } C. \]

Does \( A_C < B_D \) prove that \( A \) is better than \( B \)?

No! Beware change in CPU.

Maybe \( A_C > B_C \); \( A_D > B_D \);

\( C \) does more work per cycle than \( D \), thanks to CPU manufacturer.

Sometimes people measure cost in seconds instead of cycles.

Then they benefit from more work per cycle and from more cycles per second.

Better comparisons (still raising many questions):

ECDH on Intel Pentium II/III (still not exactly the same):

1920000 cycles for NIST P-256, 832457 cycles for Curve25519.

ECDH on Sandy Bridge:

374000 cycles for NIST P-256 (from 2013 Gueron–Krasnov), 159128 cycles for Curve25519.

Verification on Sandy Bridge:

529000 cycles for ECDSA-P-256, 205741 cycles for Ed25519.
$A_C$: cycles for alg $A$ on CPU $C$. Does $A_C < B_D$ prove that $A$ is better than $B$?

No! Beware change in CPU.

Maybe $A_C > B_C$; $A_D > B_D$; $C$ does more work per cycle than $D$, thanks to CPU manufacturer.

Sometimes people measure cost in seconds instead of cycles. Then they benefit from more work per cycle and from more cycles per second.

Better comparisons (still raising many questions):

ECDH on Intel Pentium II/III (still not exactly the same):
1920000 cycles for NIST P-256, 832457 cycles for Curve25519.

ECDH on Sandy Bridge:
374000 cycles for NIST P-256 (from 2013 Gueron–Krasnov), 159128 cycles for Curve25519.

Verification on Sandy Bridge:
529000 cycles for ECDSA-P-256, 205741 cycles for Ed25519.
A\textsubscript{C} < B\textsubscript{D} prove that A is better than B?

No! Beware change in CPU.

A\textsubscript{C} > B\textsubscript{C}; A\textsubscript{D} > B\textsubscript{D}; more work per cycle than thanks to CPU manufacturer.

Sometimes people measure cost in seconds instead of cycles. They benefit more work per cycle and more cycles per second.

Better comparisons (still raising many questions):

ECDH on Intel Pentium II/III (still not exactly the same):
1920000 cycles for NIST P-256,
832457 cycles for Curve25519.

ECDH on Sandy Bridge:
374000 cycles for NIST P-256 (from 2013 Gueron–Krasnov),
159128 cycles for Curve25519.

Verification on Sandy Bridge:
529000 cycles for ECDSA-P-256,
205741 cycles for Ed25519.

For each of these operations, on each of these curves, on each of these CPUs:

Simplest implementations are much, much, much slower.

Questions in algorithm design and software engineering:
How to build the fastest software on, e.g., an ARM Cortex-A8 for Ed25519?

Answers feed back into crypto design: choosing fast curves.
A on CPU C.

Does \( A \) on CPU C. prove that \( A < B \)?

No! Beware change in CPU.

\( A_D > B_D \); per cycle than manufacturer.

measure cost of cycles.

Better comparisons (still raising many questions):

ECDH on Intel Pentium II/III (still not exactly the same):
1920000 cycles for NIST P-256,
832457 cycles for Curve25519.

ECDH on Sandy Bridge:
374000 cycles for NIST P-256
(from 2013 Gueron–Krasnov),
159128 cycles for Curve25519.

Verification on Sandy Bridge:
529000 cycles for ECDSA-P-256,
205741 cycles for Ed25519.

Simplest implementations are much, much, much slower.

Questions in algorithm design and software engineering:

How to build the fastest software on, e.g., an ARM Cortex-A8 for Ed25519 signature verification?

Answers feed back into crypto design: e.g., choosing fast curves.
Better comparisons (still raising many questions):

ECDH on Intel Pentium II/III (still not exactly the same):
1920000 cycles for NIST P-256, 832457 cycles for Curve25519.

ECDH on Sandy Bridge:
374000 cycles for NIST P-256 (from 2013 Gueron–Krasnov), 159128 cycles for Curve25519.

Verification on Sandy Bridge:
529000 cycles for ECDSA-P-256, 205741 cycles for Ed25519.

For each of these operations, on each of these curves, on each of these CPUs:

Simplest implementations are much, much, much slower.
Questions in algorithm design and software engineering:
How to build the fastest software on, e.g., an ARM Cortex-A8 for Ed25519 signature verification?
Answers feed back into crypto design: e.g., choosing fast curves.

Does \( A \) prove that \( A \) is better than \( B \)?
No! Beware change in CPU.

Maybe \( A \) \( > B \);
\( C \) does more work per cycle, thanks to CPU manufacturer.

Sometimes people measure cost in seconds instead of cycles.
Then they benefit from more work per cycle and from more cycles per second.

For each of these operations, on each of these curves, on each of these CPUs:

Simplest implementations are much, much, much slower.
Questions in algorithm design and software engineering:
How to build the fastest software on, e.g., an ARM Cortex-A8 for Ed25519 signature verification?
Answers feed back into crypto design: e.g., choosing fast curves.
Better comparisons (still raising many questions):
ECDH on Intel Pentium II/III (still not exactly the same):
1920000 cycles for NIST P-256, 832457 cycles for Curve25519.
ECDH on Sandy Bridge:
374000 cycles for NIST P-256 (from 2013 Gueron–Krasnov),
159128 cycles for Curve25519.
Verification on Sandy Bridge:
529000 cycles for ECDSA-P-256, 205741 cycles for Ed25519.

For each of these operations, on each of these curves, on each of these CPUs:
Simplest implementations are much, much, much slower.
Questions in algorithm design and software engineering:
How to build the fastest software on, e.g., an ARM Cortex-A8 for Ed25519 signature verification?
Answers feed back into crypto design: e.g., choosing fast curves.
Better comparisons (still raising many questions):
ECDH on Intel Pentium II/III (not exactly the same):
1920000 cycles for NIST P-256,
832457 cycles for Curve25519.
ECDH on Sandy Bridge:
374000 cycles for NIST P-256
(from 2013 Gueron–Krasnov),
159128 cycles for Curve25519.
Verification on Sandy Bridge:
529000 cycles for ECDSA-P-256,
205741 cycles for Ed25519.

For each of these operations,
on each of these curves,
on each of these CPUs:
Simplest implementations are much, much, much slower.
Questions in algorithm design and software engineering:
How to build the fastest software on, e.g., an ARM Cortex-A8 for
Ed25519 signature verification?
Answers feed back into crypto design: e.g., choosing fast curves.

Several levels to optimize:
ECC ops: e.g., verify \( SB = R + hA \)
windowing etc.
\[ \downarrow \downarrow \]
Point ops: e.g., \( P;Q \mapsto P + Q \)
faster doubling etc.
\[ \downarrow \downarrow \]
Field ops: e.g., \( x_1; x_2 \mapsto x_1 x_2 \) in \( F_p \)
delayed carries etc.
\[ \downarrow \downarrow \]
Machine insns: e.g., 32-bit multiplication
\[ \downarrow \downarrow \]
pipelining etc.
\[ \downarrow \downarrow \]
Gates: e.g., AND, OR, XOR
Better comparisons (still raising many questions):
ECDH on Intel Pentium II/III (still not exactly the same):
- 1920000 cycles for NIST P-256,
- 832457 cycles for Curve25519.
ECDH on Sandy Bridge:
- 374000 cycles for NIST P-256 (from 2013 Gueron–Krasnov),
- 159128 cycles for Curve25519.
Verification on Sandy Bridge:
- 529000 cycles for ECDSA-P-256,
- 205741 cycles for Ed25519.

For each of these operations, on each of these curves, on each of these CPUs:
Simplest implementations are much, much, much slower.

Questions in algorithm design and software engineering:
How to build the fastest software on, e.g., an ARM Cortex-A8 for Ed25519 signature verification?

Answers feed back into crypto design: e.g., choosing fast curves.

Several levels to optimize:
ECC ops: e.g., verify $SB = R + hA$
\[\Downarrow\Downarrow\]
Point ops: e.g., $P, Q \mapsto P + Q$
\[\Downarrow\Downarrow\]
Field ops: e.g., $x_1, x_2 \mapsto x_1x_2$ in $\mathbb{F}_p$
\[\Downarrow\Downarrow\]
Machine insns: e.g., 32-bit multiplication
\[\Downarrow\Downarrow\]
Gates: e.g., AND, OR, XOR
For each of these operations, on each of these curves, on each of these CPUs:

Simplest implementations are much, much, much slower.

Questions in algorithm design and software engineering:
How to build the fastest software on, e.g., an ARM Cortex-A8 for Ed25519 signature verification?

Answers feed back into crypto design: e.g., choosing fast curves.

Several levels to optimize:

- **ECC ops**: e.g., verify $SB = R + hA$
  - windowing etc.

- **Point ops**: e.g., $P, Q \mapsto P + Q$
  - faster doubling etc.

- **Field ops**: e.g., $x_1, x_2 \mapsto x_1x_2$ in $\mathbb{F}_p$
  - delayed carries etc.

- **Machine insns**: e.g., 32-bit multiplication
  - pipelining etc.

- **Gates**: e.g., AND, OR, XOR
For each of these operations, on each of these curves, on each of these CPUs:

Simplest implementations are much, much, much slower.

Questions in algorithm design and software engineering:
How to build the fastest software on, e.g., an ARM Cortex-A8 for Ed25519 signature verification?

Answers feed back into crypto design: e.g., choosing fast curves.

Several levels to optimize:

- **ECC ops:** e.g., verify $SB = R + hA$
  - windowing etc.

- **Point ops:** e.g., $P, Q \mapsto P + Q$
  - faster doubling etc.

- **Field ops:** e.g., $x_1, x_2 \mapsto x_1 x_2$ in $\mathbb{F}_p$
  - delayed carries etc.

- **Machine insns:** e.g., 32-bit multiplication
  - pipelining etc.

- **Gates:** e.g., AND, OR, XOR
For each of these operations, on each of these curves, on each of these CPUs:

Simplest implementations are much, much, much slower.

Questions in algorithm design and software engineering: How to build the fastest software on, e.g., an ARM Cortex-A8 for Ed25519 signature verification?

Answers feed back into crypto design: e.g., choosing fast curves.

Several levels to optimize:

- **ECC ops**: e.g.,
  - verify $SB = R + hA$
  - windowing etc.

- **Point ops**: e.g.,
  - $P;Q \mapsto P + Q$
  - faster doubling etc.

- **Field ops**: e.g.,
  - $x_1, x_2 \mapsto x_1x_2$ in $\mathbb{F}_p$
  - delayed carries etc.

- **Machine insns**: e.g.,
  - 32-bit multiplication
  - pipelining etc.

- **Gates**: e.g.,
  - AND, OR, XOR

**Single-scalar multiplication**

Fundamental ECC operation:

$$n, P \mapsto nP$$

Input $n$ is integer in, e.g.,
\[\ldots 0; 1; \ldots\]

Input $P$ is point on elliptic curve.

Will build $n; P \mapsto nP$ using additions and subtractions $P; Q \mapsto P + Q$.

Later will also look at double-scalar multiplication:

$$m, P, n, Q \mapsto mP + nQ.$$
For each of these operations, on each of these curves, on each of these CPUs:

- Simplest implementations are much, much, much slower.
- Questions in algorithm design and software engineering:
  How to build the fastest software on, e.g., an ARM Cortex-A8 for Ed25519 signature verification?

Answers feed back into crypto design: e.g., choosing fast curves.

Several levels to optimize:

<table>
<thead>
<tr>
<th>ECC ops: e.g., verify $SB = R + hA$</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowing etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Point ops: e.g., $P, Q \rightarrow P + Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>faster doubling etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field ops: e.g., $x_1, x_2 \rightarrow x_1x_2$ in $\mathbb{F}_p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>delayed carries etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine insns: e.g., 32-bit multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipelining etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gates: e.g., AND, OR, XOR</th>
</tr>
</thead>
</table>

Single-scalar multiplication

Fundamental ECC operation: $n, P \rightarrow nP$.

Input $n$ is integer in $\{0, 1, \ldots, 2^{256} - 1\}$.

Input $P$ is point on elliptic curve.

Will build $n, P \rightarrow nP$ using additions $P, Q \rightarrow P + Q$ and subtractions $P, Q \rightarrow P - Q$.

Later will also look at double-scalar multiplication: $m, P, n, Q \rightarrow mP + nQ$. 

Single-scalar multiplication

Fundamental ECC operation: $n, P \rightarrow nP$.

Input $n$ is integer in $\{0, 1, \ldots, 2^{256} - 1\}$.

Input $P$ is point on elliptic curve.

Will build $n, P \rightarrow nP$ using additions $P, Q \rightarrow P + Q$ and subtractions $P, Q \rightarrow P - Q$.

Later will also look at double-scalar multiplication: $m, P, n, Q \rightarrow mP + nQ$.
For each of these operations, on each of these curves, on each of these CPUs:

Simplest implementations are much, much, much slower.

Questions in algorithm design and software engineering:

- How to build the fastest software on, e.g., an ARM Cortex-A8 for Ed25519 signature verification?
- Answers feed back into crypto design: e.g., choosing fast curves.

Several levels to optimize:

- **ECC ops**: e.g., verify $SB = R + hA$
  - windowing etc.
- **Point ops**: e.g., $P, Q \mapsto P + Q$
  - faster doubling etc.
- **Field ops**: e.g., $x_1, x_2 \mapsto x_1x_2$ in $\mathbb{F}_p$
  - delayed carries etc.
- **Machine insns**: e.g., 32-bit multiplication
  - pipelining etc.
- **Gates**: e.g., AND, OR, XOR

---

Single-scalar multiplication

Fundamental ECC operation:

$n, P \mapsto nP$.

Input $n$ is integer in, e.g., $\{0, 1, \ldots, 2^{256} - 1\}$.

Input $P$ is point on elliptic curve.

Will build $n, P \mapsto nP$ using additions $P, Q \mapsto P + Q$ and subtractions $P, Q \mapsto P - Q$.

Later will also look at double-scalar multiplication $m, P, n, Q \mapsto mP + nQ$. 
Several levels to optimize:

**ECC ops:** e.g., verify $SB = R + hA$
- windowing etc.

**Point ops:** e.g., $P, Q \mapsto P + Q$
- faster doubling etc.

**Field ops:** e.g., $x_1, x_2 \mapsto x_1 x_2$ in $\mathbb{F}_p$
- delayed carries etc.

**Machine insns:** e.g., 32-bit multiplication
- pipelining etc.

**Gates:** e.g., AND, OR, XOR

---

**Single-scalar multiplication**

Fundamental ECC operation: $n, P \mapsto nP$.

Input $n$ is integer in, e.g., $\{0, 1, \ldots, 2^{256} - 1\}$.

Input $P$ is point on elliptic curve.

Will build $n, P \mapsto nP$ using additions $P, Q \mapsto P + Q$
and subtractions $P, Q \mapsto P - Q$.

Later will also look at double-scalar multiplication
$m, P, n, Q \mapsto mP + nQ$. 

Single-scalar multiplication

Fundamental ECC operation:

\[ n, P \mapsto nP. \]

Input \( n \) is integer in, e.g., \( \{0, 1, \ldots, 2^{256} - 1\} \).

Input \( P \) is point on elliptic curve.

Will build \( n, P \mapsto nP \) using additions \( P, Q \mapsto P + Q \) and subtractions \( P, Q \mapsto P - Q \).

Later will also look at double-scalar multiplication \( m, P, n, Q \mapsto mP + nQ \).

Left-to-right binary method

```python
def scalarmult(n, P):
    if n == 0: return 0
    if n == 1: return P
    R = scalarmult(n//2, P)
    R = R + R
    if n % 2: R = R + P
    return R
```

Two Python notes:

- \( n//2 \) in Python means \( \lfloor n / 2 \rfloor \).
- Recursion depth is limited.
  See `sys.setrecursionlimit`.
Several levels to optimize:

ECC ops: e.g.,
\(\text{verify } SB = R + hA\)
windowing etc.

\[\downarrow \quad \downarrow\]

Point ops: e.g.,
\(P;Q \mapsto P + Q\)
faster doubling etc.

\[\downarrow \quad \downarrow\]

Field ops: e.g.,
\(x_1; x_2 \mapsto x_1 x_2\)
in \(\mathbb{F}_p\)
delayed carries etc.

\[\downarrow \quad \downarrow\]

Machine insns: e.g.,
32-bit multiplication
\[\downarrow \quad \downarrow\]
pipelining etc.

\[\downarrow \quad \downarrow\]

Gates: e.g.,
AND, OR, XOR

---

Single-scalar multiplication

Fundamental ECC operation:
\(n, P \mapsto nP\).

Input \(n\) is integer in, e.g.,
\(\{0, 1, \ldots, 2^{256} - 1\}\).

Input \(P\) is point on elliptic curve.

Will build \(n, P \mapsto nP\)
using additions \(P, Q \mapsto P + Q\)
and subtractions \(P, Q \mapsto P - Q\).

Later will also look at
double-scalar multiplication
\(m, P, n, Q \mapsto mP + nQ\).

---

Left-to-right binary method

```python
def scalarmult(n, P):
    if n == 0:
        return 0
    if n == 1:
        return P
    R = scalarmult(n//2, P)
    R = R + R
    if n % 2:
        R = R + P
    return R
```

Two Python notes:

- \(n//2\) in Python means \(\lfloor n/2 \rfloor\).
- Recursion depth is limited.

See `sys.setrecursionlimit`.
Single-scalar multiplication

Fundamental ECC operation:
\( n, P \mapsto nP \).

Input \( n \) is integer in, e.g., \( \{0, 1, \ldots, 2^{256} - 1\} \).

Input \( P \) is point on elliptic curve.

Will build \( n, P \mapsto nP \)
using additions \( P, Q \mapsto P + Q \)
and subtractions \( P, Q \mapsto P - Q \).

Later will also look at
double-scalar multiplication
\( m, P, n, Q \mapsto mP + nQ \).

Left-to-right binary method

```python
def scalarmult(n,P):
    if n == 0: return 0
    if n == 1: return P
    R = scalarmult(n//2,P)
    R = R + R
    if n % 2: R = R + P
    return R
```

Two Python notes:

- \( n//2 \) in Python means \( \lfloor n/2 \rfloor \).
- Recursion depth is limited.

See `sys.setrecursionlimit`.
Single-scalar multiplication

Fundamental ECC operation:
\[ n, P \mapsto nP. \]

Input \( n \) is integer in, e.g., \( \{0, 1, \ldots, 2^{256} - 1 \} \).

Input \( P \) is point on elliptic curve.

Will build \( n, P \mapsto nP \)
using additions \( P, Q \mapsto P + Q \)
and subtractions \( P, Q \mapsto P - Q \).

Later will also look at
double-scalar multiplication
\[ m, P, n, Q \mapsto mP + nQ. \]

Left-to-right binary method

\[
def \text{scalarmult}(n, P):
    \begin{align*}
    &\text{if } n == 0: \text{return } 0 \\
    &\text{if } n == 1: \text{return } P \\
    &R = \text{scalarmult}(n//2, P) \\
    &R = R + R \\
    &\text{if } n \% 2: R = R + P \\
    &\text{return } R
    \end{align*}
\]

Two Python notes:

- \( n//2 \) in Python means \( \lfloor n/2 \rfloor \).
- Recursion depth is limited. See \text{sys.setrecursionlimit}. 

Single-scalar multiplication

Fundamental ECC operation:

\[ n;P \mapsto nP . \]

Input \( n \) is integer in, e.g., \( 0, 1, \ldots, 2^{256} - 1 \). \( n \) is point on elliptic curve.

We define \n, \( P \mapsto nP \)

additions \( P, Q \mapsto P + Q \)

subtractions \( P, Q \mapsto P - Q \).

We will also look at

scalar multiplication \( Q \mapsto mP + nQ \).

Left-to-right binary method

```python
def scalarmult(n, P):
    if n == 0: return 0
    if n == 1: return P
    R = scalarmult(n//2, P)
    R = R + R
    if n % 2: R = R + P
    return R
```

Two Python notes:

- \( n//2 \) in Python means \( \lfloor n/2 \rfloor \).
- Recursion depth is limited.

See `sys.setrecursionlimit`.

This recursion computes \( nP \) as

- \( 2^{\lceil n \rceil} / 2 P \)
e.g. \( 20 P = 2 \cdot 10 P \).
- \( 2^{\lceil n - 1 \rceil} / 2 P \) + \( P \)
e.g. \( 21 P = 2 \cdot 10 P + P \).

Base cases in recursion:

\( 0P = 0 \).
\( 1P = P \). Could omit this case.

Assuming \( n \geq 0 \) for simplicity.

Otherwise use \( nP = \ominus (\ominus n) P \).
Single-scalar multiplication

Fundamental ECC operation: \(n; P \mapsto nP\).

Input \(n\) is integer in, e.g., \(\mathbb{Z} = 0; 1; \ldots; 2^{256} - 1\).

Input \(P\) is point on elliptic curve.

Will build \(n; P \mapsto nP\) using additions \(P; Q \mapsto P + Q\) and subtractions \(P; Q \mapsto P - Q\).

Later will also look at double-scalar multiplication \(m; P; n; Q \mapsto mP + nQ\).

Left-to-right binary method

```
def scalarmult(n, P):
    if n == 0: return 0
    if n == 1: return P
    R = scalarmult(n//2, P)
    R = R + R
    if n % 2: R = R + P
    return R
```

Two Python notes:
• \(n//2\) in Python means \([n/2]\).
• Recursion depth is limited. See `sys.setrecursionlimit`.

This recursion computes \(nP\) as

- \(2 \left( \frac{n}{2} P \right)\) if \(n \in 2\mathbb{Z}\), e.g. \(20P = 2 \cdot 10P\).
- \(2 \left( \frac{n-1}{2} P \right) + P\) if \(n \in 1 + 2\mathbb{Z}\), e.g. \(21P = 2 \cdot 10P + P\).

Base cases in recursion:
- \(0P = 0\). For Edwards: \(0 = (0; 1)\).
- \(1P = P\). Could omit this case.

Assuming \(n \geq 0\) for simplicity.
Otherwise use \(nP = -(-n)P\).
Single-scalar multiplication

Fundamental ECC operation:

\[ n; P \mapsto nP \].

Input \( n \) is integer in, e.g.,

\[ \overline{0}; 1; \ldots; 2^{256} - 1 \].

Input \( P \) is point on elliptic curve.

Will build \( n; P \mapsto nP \) using additions \( P; Q \mapsto P + Q \) and subtractions \( P; Q \mapsto P - Q \).

Later will also look at double-scalar multiplication

\[ m; P; n; Q \mapsto mP + nQ . \]

---

Left-to-right binary method

def scalarmult(n,P):
    if n == 0: return 0
    if n == 1: return P
    R = scalarmult(n//2,P)
    R = R + R
    if n % 2: R = R + P
    return R

Two Python notes:

- \( n//2 \) in Python means \( \lfloor n/2 \rfloor \).
- Recursion depth is limited.
  See \texttt{sys.setrecursionlimit}.

This recursion computes \( nP \) as

- \( 2 \left( \frac{n}{2} \right) \) if \( n \in 2\mathbb{Z} \).
  e.g. \( 20P = 2 \cdot 10P \).

- \( 2 \left( \frac{n-1}{2} \right) + P \) if \( n \in 1 + 2\mathbb{Z} \).
  e.g. \( 21P = 2 \cdot 10P + P \).

Base cases in recursion:

\( 0P = 0 \). For Edwards: \( 0 = (0; 1) \).
\( 1P = P \). Could omit this case.

Assuming \( n \geq 0 \) for simplicity.
Otherwise use \( nP = -(\neg n)P \).
Left-to-right binary method

def scalarmult(n,P):
    if n == 0: return 0
    if n == 1: return P
    R = scalarmult(n//2,P)
    R = R + R
    if n % 2: R = R + P
    return R

Two Python notes:
• \( n/2 \) in Python means \( \lfloor n/2 \rfloor \).
• Recursion depth is limited.
  See \texttt{sys.setrecursionlimit}.

This recursion computes \( nP \) as

- \( 2 \left( \frac{n}{2} P \right) \) if \( n \in 2\mathbb{Z} \).
  e.g. \( 20P = 2 \cdot 10P \).
- \( 2 \left( \frac{n-1}{2} P \right) + P \) if \( n \equiv 1 + 2\mathbb{Z} \).
  e.g. \( 21P = 2 \cdot 10P + P \).

Base cases in recursion:
\( 0P = 0 \). For Edwards: \( 0 = (0, 1) \).
\( 1P = P \). Could omit this case.

Assuming \( n \geq 0 \) for simplicity.
Otherwise use \( nP = -(-n)P \).
```python
def scalarmult(n, P):
    if n == 0: return 0
    if n == 1: return P
    R = scalarmult(n // 2, P)
    R = R + R
    if n % 2: R = R + P
    return R
```

This recursion computes $nP$ as

- $2 \left( \frac{n}{2} P \right)$ if $n \in 2\mathbb{Z}$.
  - e.g. $20P = 2 \cdot 10P$.
- $2 \left( \frac{n - 1}{2} P \right) + P$ if $n \in 1 + 2\mathbb{Z}$.
  - e.g. $21P = 2 \cdot 10P + P$.

Base cases in recursion:
- $0P = 0$. For Edwards: $0 = (0, 1)$.
- $1P = P$. Could omit this case.

Assuming $n \geq 0$ for simplicity.

Otherwise use $nP = -(−n)P$.

If $0 \leq n < 2^b$ then this algorithm uses $\leq 2b - 2$ additions: specifically $\leq b - 1$ doublings and $\leq b - 1$ additions of $P$.

Example of worst case:

- $31P = 2 \cdot 2 \cdot 2 \cdot 2 \cdot (2P + P) + P$.
- $31 = (11111)_2$; $b = 5$; $4$ doublings; $4$ more additions.

Average case is better: e.g.

- $35P = 2 \cdot 2 \cdot 2 \cdot 2 \cdot (2P + P) + P$.
- $35 = (100011)_2$; $b = 6$; $5$ doublings; $2$ additions.
This recursion computes $nP$ as

- $2 \left( \frac{n}{2} P \right)$ if $n \in 2\mathbb{Z}$.
  - e.g. $20P = 2 \cdot 10P$.
- $2 \left( \frac{n-1}{2} P \right) + P$ if $n \in 1 + 2\mathbb{Z}$.
  - e.g. $21P = 2 \cdot 10P + P$.

Base cases in recursion:
- $0P = 0$. For Edwards: $0 = (0,1)$.
- $1P = P$. Could omit this case.

Assuming $n \geq 0$ for simplicity. Otherwise use $nP = -(-n)P$.

If $0 \leq n < 2^b$ then this algorithm uses $\leq 2^b - 2$ additions: specifically $\leq b - 1$ doublings and $\leq b - 1$ additions of $P$.

Example of worst case:
$31 = (11111)_2$; $b = 5$; 4 doublings; 4 more additions.

Average case is better: e.g.
$35 = (100011)_2$; $b = 6$; 5 doublings; 2 additions.
This recursion computes \( nP \) as

- \( 2 \left( \frac{n}{2} P \right) \) if \( n \in 2\mathbb{Z} \).
  
  e.g. \( 20P = 2 \cdot 10P \).

- \( 2 \left( \frac{n-1}{2} P \right) + P \) if \( n \in 1 + 2\mathbb{Z} \).
  
  e.g. \( 21P = 2 \cdot 10P + P \).

Base cases in recursion:

\( 0P = 0 \). For Edwards: \( 0 = (0, 1) \).

\( 1P = P \). Could omit this case.

Assuming \( n \geq 0 \) for simplicity.

Otherwise use \( nP = -(−n)P \).

If \( 0 \leq n < 2^b \) then
this algorithm uses
\( \leq 2b - 2 \) additions: specifically
\( \leq b - 1 \) doublings and
\( \leq b - 1 \) additions of \( P \).

Example of worst case:

\( 31P = 2(2(2(2P + P) + P) + P) + P \).
\( 31 = (11111)_2; b = 5; \)
4 doublings; 4 more additions.

Average case is better: e.g.

\( 35P = 2(2(2(2(2P))) + P) + P \).
\( 35 = (100011)_2; b = 6; \)
5 doublings; 2 additions.
This recursion computes \(nP\) as

- \(2 \left( \frac{n}{2} \right) P\) if \(n \in 2 \mathbb{Z}\).
  
  e.g. \(20P = 2 \cdot 10P\).

- \(2 \left( \frac{n-1}{2} \right) + P\) if \(n \in 1 + 2 \mathbb{Z}\).
  
  e.g. \(21P = 2 \cdot 10P + P\).

Base cases in recursion:

\(0P = 0\). For Edwards: \(0 = (0, 1)\).

\(1P = P\). Could omit this case.

Assuming \(n \geq 0\) for simplicity.

Otherwise use \(nP = -(-n)P\).

If \(0 \leq n < 2^b\) then this algorithm uses 
\(\leq 2b - 2\) additions: specifically
\(\leq b - 1\) doublings and 
\(\leq b - 1\) additions of \(P\).

Example of worst case:

\(31P = 2(2(2(2P + P) + P) + P) + P\).

\(31 = (11111)_{2}; b = 5\);

4 doublings; 4 more additions.

Average case is better: e.g.

\(35P = 2(2(2(2P)) + P) + P\).

\(35 = (100011)_{2}; b = 6\);

5 doublings; 2 additions.
This recursion computes $nP$ as

- $2^n P$ if $n \in 2\mathbb{Z}$.
- $2^{n-1}P + P$ if $n \in 1 + 2\mathbb{Z}$.

\[ 20P = 2 \cdot 10P. \]
\[ 21P = 2 \cdot 10P + P. \]

Base cases in recursion:
- For Edwards: $0 = (0; 1)$. Could omit this case.

Assuming $n \geq 0$ for simplicity.

Otherwise use $nP = -(-n)P$.

If $0 \leq n < 2^b$ then this algorithm uses \leq 2b - 2 additions: specifically \leq b - 1 doublings and \leq b - 1 additions of $P$.

Example of worst case:
\[ 31P = 2(2(2(2P + P) + P) + P) + P. \]
\[ 31 = (11111)_2; b = 5; \]
4 doublings; 4 more additions.

Average case is better: e.g.
\[ 35P = 2(2(2(2(2P)))) + P) + P. \]
\[ 35 = (100011)_2; b = 6; \]
5 doublings; 2 additions.

Non-adjacent form (NAF)
def scalarmult(n,P):
if n == 0: return 0
if n == 1: return P
if n % 4 == 1:
R = scalarmult((n-1)/4,P)
R = R + R
return (R + R) + P
if n % 4 == 3:
R = scalarmult((n+1)/4,P)
R = R + R
return (R + R) - P
R = scalarmult(n/2,P)
return R + R
This recursion computes $nP$ as

- $2^nP$ if $n \in 2\mathbb{Z}$.
- $0P + P$ if $n \in 1 + 2\mathbb{Z}$.

E.g. $20P = 2 \cdot 10P$.

Base cases in recursion:

- $0P = 0$. For Edwards: $0 = (0; 1)$.
- $1P = P$. Could omit this case.

Assuming $n \geq 0$ for simplicity.

Otherwise use $nP = -(-n)P$.

If $0 \leq n < 2^b$ then this algorithm uses

- $\leq 2b - 2$ additions: specifically
- $\leq b - 1$ doublings and
- $\leq b - 1$ additions of $P$.

Example of worst case:


$31 = (11111)_2$; $b = 5$;

4 doublings; 4 more additions.

Average case is better: e.g.

$35P = 2(2(2(2(2P))) + P) + P$.

$35 = (100011)_2$; $b = 6$;

5 doublings; 2 additions.

Non-adjacent form (NAF)

```python
def scalarmult(n, P):
    if n == 0: return 0
    if n == 1: return P
    if n % 4 == 1:
        R = scalarmult((n-1)/4, P)
        R = R + R
        return (R + R) + P
    if n % 4 == 3:
        R = scalarmult((n+1)/4, P)
        R = R + R
        return (R + R) - P
    R = scalarmult(n/2, P)
    return R + R
```
This recursion computes $nP$ as

- $2^n \cdot P$ if $n \in 2\mathbb{Z}$.
  
  e.g. $20P = 2 \cdot 10P$.

- $2^{n-1}P \cdot P$ if $n \in 1 + 2\mathbb{Z}$.
  
  e.g. $21P = 2 \cdot 10P + P$.

Base cases in recursion:

- $0P = 0$. For Edwards: $0 = (0; 1)$.
- $1P = P$. Could omit this case.

Assuming $n \geq 0$ for simplicity.

Otherwise use $nP = -(−n)P$.

---

If $0 \leq n < 2^b$ then this algorithm uses

$\leq 2b - 2$ additions: specifically

$\leq b - 1$ doublings and

$\leq b - 1$ additions of $P$.

Example of worst case:


$31 = (11111)_2$; $b = 5$;

4 doublings; 4 more additions.

Average case is better: e.g.

$35P = 2(2(2(2P))) + P$.

$35 = (100011)_2$; $b = 6$;

5 doublings; 2 additions.

---

Non-adjacent form (NAF)

```python
def scalarmult(n,P):
    if n == 0: return 0
    if n == 1: return P
    if n % 4 == 1:
        R = scalarmult((n-1)/4,P)
        R = R + R
        return (R + R) + P
    if n % 4 == 3:
        R = scalarmult((n+1)/4,P)
        R = R + R
        return (R + R) - P
    R = scalarmult(n/2,P)
    return R + R
```
If $0 \leq n < 2^b$ then
this algorithm uses
$\leq 2b - 2$ additions: specifically
$\leq b - 1$ doublings and
$\leq b - 1$ additions of $P$.

Example of worst case:
$31 = (11111)_2; b = 5$;
4 doublings; 4 more additions.

Average case is better: e.g.
$35P = 2(2(2(2(2P)))) + P$.
$35 = (100011)_2; b = 6$;
5 doublings; 2 additions.

Non-adjacent form (NAF)

def scalarmult(n,P):
    if n == 0: return 0
    if n == 1: return P
    if n % 4 == 1:
        R = scalarmult((n-1)/4,P)
        R = R + R
        return (R + R) + P
    if n % 4 == 3:
        R = scalarmult((n+1)/4,P)
        R = R + R
        return (R + R) - P
    R = scalarmult(n/2,P)
    return R + R
If $0 \leq n < 2^b$ then this algorithm uses $\leq 2^b - 2$ additions: specifically $\leq b - 1$ doublings and $\leq b - 1$ additions of $P$.

Example of worst case:
$31P = 2(2(2(2(2P + P) + P) + P) + P)$
$31 = (11111)_2$; $b = 5$;
4 doublings; 4 more additions.

Average case is better: e.g.
$35P = 2(2(2(2(2P ))) + P) - P$
$35 = (100011)_2$; $b = 6$;
5 doublings; 2 additions.

Subtraction on the curve is as cheap as addition. NAF takes advantage of this.

“Non-adjacent”: $\pm P$ ops are separated by $\geq 2$ doublings.
Worst case: $\approx b$ doublings plus $\approx b = 2$ additions of $\pm P$.
On average $\approx b = 3$ additions.

Non-adjacent form (NAF)

def scalarmult(n,P):
    if n == 0: return 0
    if n == 1: return P
    if n % 4 == 1:
        R = scalarmult((n-1)/4,P)
        R = R + R
        return (R + R) + P
    if n % 4 == 3:
        R = scalarmult((n+1)/4,P)
        R = R + R
        return (R + R) - P
    R = scalarmult(n/2,P)
    return R + R
If \( 0 \leq n < 2^b \) then this algorithm uses \( \leq 2^b - 2 \) additions: specifically \( \leq b - 1 \) doublings and \( \leq b - 1 \) additions of \( P \).

Example of worst case:

\[ 31P = 2(2(2(2P)) + P) + P. \]

\[ 31 = (10000\bar{1})_2; \bar{1} \text{ denotes } -1. \]

\[ 35P = 2(2(2(2P))) + P. \]

\[ 35 = (10010\bar{1})_2. \]

“Non-adjacent”: \( \pm P \) ops are separated by \( \geq 2 \) doublings.

Worst case: \( \approx b \) doublings plus \( \approx b/2 \) additions.

On average \( \approx b/3 \) additions.
**Non-adjacent form (NAF)**

```python
def scalarmult(n,P):
    if n == 0: return 0
    if n == 1: return P
    if n % 4 == 1:
        R = scalarmult((n-1)/4,P)
        R = R + R
        return (R + R) + P
    if n % 4 == 3:
        R = scalarmult((n+1)/4,P)
        R = R + R
        return (R + R) - P
    R = scalarmult(n/2,P)
    return R + R
```

Subtraction on the curve is as cheap as addition. NAF takes advantage of this.

\[
31P = 2(2(2(2P))) - P. 
\]
\[
31 = (10000\overline{1})_2; \overline{1} \text{ denotes } -1.
\]
\[
35P = 2(2(2(2P)) + P) - P. 
\]
\[
35 = (10010\overline{1})_2.
\]

“Non-adjacent”: ±P ops are separated by ≥2 doublings.

Worst case: \(\approx b\) doublings plus \(\approx b/2\) additions of ±P.

On average \(\approx b/3\) additions.
Non-adjacent form (NAF)

def scalarmult(n,P):
    if n == 0: return 0
    if n == 1: return P
    if n % 4 == 1:
        R = scalarmult((n-1)/4,P)
        R = R + R
        return (R + R) + P
    if n % 4 == 3:
        R = scalarmult((n+1)/4,P)
        R = R + R
        return (R + R) - P
    R = scalarmult(n/2,P)
    return R + R

Subtraction on the curve is as cheap as addition.
NAF takes advantage of this.

31P = 2(2(2(2(2P)))) - P.
31 = (10000\overline{1})_2; \overline{1} denotes -1.

35P = 2(2(2(2(2P)) + P)) - P.
35 = (10010\overline{1})_2.

“Non-adjacent”: ±P ops are separated by ≥ 2 doublings.

Worst case: \approx b doublings plus \approx b/2 additions of ±P.
On average \approx b/3 additions.
Non-adjacent form (NAF)

```python
def scalarmult(n, P):
    if n == 0:
        return 0
    if n == 1:
        return P
    if n % 4 == 1:
        R = scalarmult((n-1)/4, P)
        R = R + R
        return (R + R) + P
    if n % 4 == 3:
        R = scalarmult((n+1)/4, P)
        R = R + R
        return (R + R) - P
    R = scalarmult(n/2, P)
    return R + R
```

Subtraction on the curve is as cheap as addition. NAF takes advantage of this.

\[ 31P = 2\left(2\left(2\left(2\left(2P\right)\right)\right)\right) - P. \]

\[ 31 = (10000\bar{1})_2; \bar{1} \text{ denotes } -1. \]

\[ 35P = 2\left(2\left(2\left(2\left(2P\right) + P\right)\right) - P. \]

\[ 35 = (10010\bar{1})_2. \]

“Non-adjacent”: \( \pm P \) ops are separated by \( \geq 2 \) doublings.

Worst case: \( \approx b \) doublings plus \( \approx b/2 \) additions of \( \pm P \).

On average \( \approx b/3 \) additions.

---

Width-2 signed sliding windows

```python
def window2(n, P, P3):
    if n == 0:
        return 0
    if n == 1:
        return P
    if n == 3:
        return P3
    if n % 8 == 1:
        R = window2((n-1)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) + P
    if n % 8 == 3:
        R = window2((n-3)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) + P3
```

---
Subtraction on the curve is as cheap as addition. NAF takes advantage of this.

\[ 31P = 2(2(2(2P))) - P. \]
\[ 31 = (10000\overline{1})_2; \overline{1} \text{ denotes } -1. \]

\[ 35P = 2(2(2(2P) + P)) - P. \]
\[ 35 = (10010\overline{1})_2. \]

“Non-adjacent”: \( \pm P \) ops are separated by \( \geq 2 \) doublings.

Worst case: \( \approx b \) doublings plus \( \approx b/2 \) additions of \( \pm P \). On average \( \approx b/3 \) additions.
Subtraction on the curve is as cheap as addition. NAF takes advantage of this.

\[ 31P = 2(2(2(2P))) - P. \]
\[ 31 = (10000\overline{1})_2; \overline{1} \text{ denotes } -1. \]

\[ 35P = 2(2(2(2P) + P)) - P. \]
\[ 35 = (10010\overline{1})_2. \]

“Non-adjacent”: ±P ops are separated by ≥2 doublings.

Worst case: \( \approx b \) doublings plus \( \approx b/2 \) additions of ±P.

On average \( \approx b/3 \) additions.

---

Width-2 signed sliding windows

```python
def window2(n,P,P3):
    if n == 0: return 0
    if n == 1: return P
    if n == 3: return P3
    if n % 8 == 1:
        R = window2((n-1)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) + P
    if n % 8 == 3:
        R = window2((n-3)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) + P3
```
Subtraction on the curve is as cheap as addition. NAF takes advantage of this.

$$31P = 2(2(2(2(2 P)))) - P.$$  
$$31 = (10000\bar{1})_2; \bar{1} \text{ denotes } -1.$$  

$$35P = 2(2(2(2P)) + P)) - P.$$  
$$35 = (10010\bar{1})_2.$$  

“Non-adjacent”: \(\pm P\) ops are separated by \(\geq 2\) doublings.

Worst case: \(\approx b\) doublings plus \(\approx b/2\) additions of \(\pm P\).

On average \(\approx b/3\) additions.

---

Width-2 signed sliding windows

def window2(n,P,P3):
    if n == 0: return 0
    if n == 1: return P
    if n == 3: return P3
    if n % 8 == 1:
        R = window2((n-1)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) + P
    if n % 8 == 3:
        R = window2((n-3)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) + P3
Subtraction on the curve is as cheap as addition. NAF takes advantage of this.

\[ 31P = 2(2(2(2(2P)))) - P. \]

\[ 31 = (10001\bar{1})_2; \bar{1} \text{ denotes } -1. \]

"Non-adjacent": ±P ops are separated by ≥2 doublings.

Worst case: \( \approx b \) doublings plus \( \approx b/2 \) additions of ±P.

On average \( \approx b/3 \) additions.

---

**Width-2 signed sliding windows**

```python
def window2(n, P, P3):
    if n == 0:
        return 0
    if n == 1:
        return P
    if n == 3:
        return P3
    if n % 8 == 1:
        R = window2((n-1)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) + P
    if n % 8 == 3:
        R = window2((n-3)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) + P3
    if n % 8 == 5:
        R = window2((n+3)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) - P3
    if n % 8 == 7:
        R = window2((n+1)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) - P
    R = window2(n/2, P, P3)
    return R + R

def scalarmult(n, P):
    return window2(n, P, P+P+P)
```

---
Subtraction on the curve is as cheap as addition. NAF takes advantage of this.

$$31 \cdot P = 2(2(2(2(2 \cdot P)))) - P.$$  

$$31 = (10000 \bar{1})_2$$ denotes $$-1.$$  

$$35 \cdot P = 2(2(2(2(2 \cdot P))) + P) - P.$$  

$$35 = (10010 \bar{1})_2.$$  

"Non-adjacent": $$\pm P$$ ops are separated by $$\geq 2$$ doublings.

Worst case: $$\approx b$$ doublings plus $$\approx b = 2$$ additions of $$\pm P.$$  

On average $$\approx b = 3$$ additions.

---

### Width-2 signed sliding windows

```python
def window2(n, P, P3):
    if n == 0: return 0
    if n == 1: return P
    if n == 3: return P3
    if n % 8 == 1:
        R = window2((n-1)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) + P
    if n % 8 == 3:
        R = window2((n-3)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) + P3
    if n % 8 == 5:
        R = window2((n+3)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) - P3
    if n % 8 == 7:
        R = window2((n+1)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) - P
    R = window2(n/2, P, P3)
    return R + R
```

```python
def scalarmult(n, P):
    return window2(n, P, P+P+P)
```
Subtraction on the curve is as cheap as addition. NAF takes advantage of this.

$$31 P = 2(2(2(2(2P))) - P)$$

$$31 = (10000 \overline{1})_2; \overline{1} \text{ denotes } ^{-1}.$$

$$35 P = 2(2(2(2(2P)) + P) - P).$$

$$35 = (10010 \overline{1})_2.$$

"Non-adjacent": $\pm P$ ops are separated by $\geq 2$ doublings.

Worst case: $\approx b$ doublings plus $\approx b/2$ additions of $\pm P$.

On average $\approx b/3$ additions.

---

**Width-2 signed sliding windows**

```python
def window2(n,P,P3):
    if n == 0: return 0
    if n == 1: return P
    if n == 3: return P3
    if n % 8 == 1:
        R = window2((n-1)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) + P
    if n % 8 == 3:
        R = window2((n-3)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) + P3
    if n % 8 == 5:
        R = window2((n+3)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) - P3
    if n % 8 == 7:
        R = window2((n+1)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) - P
    R = window2(n/2,P,P3)
    return R + R
```

```python
def scalarmult(n,P):
    return window2(n,P,P+P+P)
```

```python
if n % 8 == 5:
    R = window2((n+3)/8,P,P3)
    R = R + R
    R = R + R
    return (R + R) - P3
if n % 8 == 7:
    R = window2((n+1)/8,P,P3)
    R = R + R
    R = R + R
    return (R + R) - P
R = window2(n/2,P,P3)
return R + R
```
Width-2 signed sliding windows

def window2(n,P,P3):
    if n == 0: return 0
    if n == 1: return P
    if n == 3: return P3
    if n % 8 == 1:
        R = window2((n-1)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) + P
    if n % 8 == 3:
        R = window2((n-3)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) + P3
    if n % 8 == 5:
        R = window2((n+3)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) - P3
    if n % 8 == 7:
        R = window2((n+1)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) - P
    R = window2(n/2,P,P3)
    return R + R

def scalarmult(n,P):
    return window2(n,P,P+P+P)
Width-2 signed sliding windows

```python
def window2(n, P, P3):
    if n == 0: return 0
    if n == 1: return P
    if n == 3: return P3
    if n % 8 == 1:
        R = window2((n-1)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) + P
    if n % 8 == 3:
        R = window2((n-3)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) + P3
    if n % 8 == 5:
        R = window2((n+3)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) - P3
    if n % 8 == 7:
        R = window2((n+1)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) - P
    R = window2(n/2, P, P3)
    return R + R
```

def scalarmult(n, P):
    return window2(n, P, P+P+P)
```

Worst case: \( \approx b/3 \) additions;
On average \( \approx b/4 \) additions.
Width-2 signed sliding windows

```python
def window2(n, P, P3):
    if n == 0:
        return 0
    if n == 1:
        return P
    if n == 3:
        return P3
    if n % 8 == 1:
        R = window2((n-1)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) + P
    if n % 8 == 3:
        R = window2((n-3)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) + P3
    if n % 8 == 5:
        R = window2((n+3)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) - P3
    if n % 8 == 7:
        R = window2((n+1)/8, P, P3)
        R = R + R
        R = R + R
        return (R + R) - P
    return window2(n/2, P, P3)

def scalarmult(n, P):
    return window2(n, P, P+P+P)
```

Worst case: $\approx b$ doublings plus $\approx b/3$ additions of $\pm P$ or $\pm 3P$.

On average $\approx b/4$ additions.
def window2(n,P,P3):
    if n == 0: return 0
    if n == 1: return P
    if n == 3: return P3
    if n % 8 == 1:
        R = window2((n-1)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) + P
    if n % 8 == 3:
        R = window2((n-3)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) + P3
    if n % 8 == 5:
        R = window2((n+3)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) - P3
    if n % 8 == 7:
        R = window2((n+1)/8,P,P3)
        R = R + R
        R = R + R
        return (R + R) - P
    R = window2(n/2,P,P3)
    return R + R

def scalarmult(n,P):
    return window2(n,P,P+P+P)

Worst case: \( \approx b \) doublings plus \( \approx b/3 \) additions of \( \pm P \) or \( \pm 3P \).
On average \( \approx b/4 \) additions.
if n % 8 == 5:
    R = window2((n+3)/8,P,P3)
    R = R + R
    R = R + R
    return (R + R) - P3
if n % 8 == 7:
    R = window2((n+1)/8,P,P3)
    R = R + R
    R = R + R
    return (R + R) - P
R = window2(n/2,P,P3)
return R + R

def scalarmult(n,P):
    return window2(n,P,P+P+P)

Worst case: \( \approx b \) doublings plus
\( \approx b/3 \) additions of \( \pm P \) or \( \pm 3P \).
On average \( \approx b/4 \) additions.
if n % 8 == 5:
    R = window2((n+3)/8,P,P3)
    R = R + R
    R = R + R
    return (R + R) - P3
if n % 8 == 7:
    R = window2((n+1)/8,P,P3)
    R = R + R
    R = R + R
    return (R + R) - P
R = window2(n/2,P,P3)
return R + R

def scalarmult(n,P):
    return window2(n,P,P+P+P)

Worst case: $\approx b$ doublings plus $\approx b/3$ additions of $\pm P$ or $\pm 3P$.
On average $\approx b/4$ additions.

Width-3 signed sliding windows:
Precompute $P, 3P, 5P, 7P$.
On average $\approx b/5$ additions.
if n % 8 == 5:
    R = window2((n+3)/8,P,P3)
    R = R + R
    R = R + R
    return (R + R) - P3
if n % 8 == 7:
    R = window2((n+1)/8,P,P3)
    R = R + R
    R = R + R
    return (R + R) - P
R = window2(n/2,P,P3)
return R + R

def scalarmult(n,P):
    return window2(n,P,P+P+P)

Worst case: $\approx b$ doublings plus $\approx b/3$ additions of $\pm P$ or $\pm 3P$. On average $\approx b/4$ additions.

Width-3 signed sliding windows:
Precompute $P, 3P, 5P, 7P$. On average $\approx b/5$ additions.

if n % 8 == 5:
    R = window2((n+3)/8,P,P3)
    R = R + R
    R = R + R
    return (R + R) - P3
if n % 8 == 7:
    R = window2((n+1)/8,P,P3)
    R = R + R
    R = R + R
    return (R + R) - P
R = window2(n/2,P,P3)
return R + R

def scalarmult(n,P):
    return window2(n,P,P+P+P)

Worst case: \( \approx b \) doublings plus
\( \approx b/3 \) additions of \( \pm P \) or \( \pm 3P \). On average \( \approx b/4 \) additions.

Width-3 signed sliding windows:
Precompute \( P, 3P, 5P, 7P \).
On average \( \approx b/5 \) additions.

On average \( \approx b/6 \) additions.

Cost of precomputation eventually outweighs savings.
Optimal: \( \approx b \) doublings plus roughly \( b/\lg b \) additions.
if $n \% 8 == 5$:
    R = window2((n+3)/8, P, P3)
    R = R + R
    R = R + R
    return (R + R) - P3
if $n \% 8 == 7$:
    R = window2((n+1)/8, P, P3)
    R = R + R
    R = R + R
    return (R + R) - P
R = window2(n/2, P, P3)
return R + R

def scalarmult(n, P):
    return window2(n, P, P+P+P)

Worst case: $\approx b$ doublings plus $\approx b/3$ additions of $\pm P$ or $\pm 3P$. On average $\approx b/4$ additions.

Width-3 signed sliding windows:
Precompute $P, 3P, 5P, 7P$.
On average $\approx b/5$ additions.

Width 4: Precompute
On average $\approx b/6$ additions.

Cost of precomputation eventually outweighs savings.
Optimal: $\approx b$ doublings plus roughly $b/\lg b$ additions.

Double-scalar multiplication
Want to quickly compute $m; P; n; Q \mapsto mP + nQ$.
e.g. verify signature $(R; S)$ by computing $h = H(R; M)$, computing $SB - hA$, checking whether $R = SB - hA$.

Obvious approach: Compute $mP$; compute $nQ$; add.
e.g. $b = 256$:
$\approx 256$ doublings for $mP$,
$\approx 256$ doublings for $nQ$,
$\approx 50$ additions for $mP$,
$\approx 50$ additions for $nQ$. 
Worst case: $\approx b$ doublings plus $\approx b/3$ additions of $\pm P$ or $\pm 3P$.
On average $\approx b/4$ additions.

Width-3 signed sliding windows:
Precompute $P, 3P, 5P, 7P$.
On average $\approx b/5$ additions.

Width 4:
On average $\approx b/6$ additions.

Cost of precomputation eventually outweighs savings.

Optimal: $\approx b$ doublings plus roughly $b/\lg b$ additions.

Double-scalar multiplication
Want to quickly compute $m, P, n, Q \mapsto mP + nQ$.
E.g. verify signature $(R, S)$ by computing $h = H(R, M)$, computing $SB - hA$, checking whether $R = SB - hA$.

Obvious approach:
Compute $mP$; compute $nQ$; add.
E.g. $b = 256$:
$\approx 256$ doublings for $mP$,
$\approx 256$ doublings for $nQ$,
$\approx 50$ additions for $mP$,
$\approx 50$ additions for $nQ$.
Double-scalar multiplication

Want to quickly compute $m, P, n, Q \mapsto mP + nQ$.

e.g. verify signature $(R, S)$ by computing $h = H(R, M)$, computing $SB - hA$, checking whether $R = SB - hA$.

Obvious approach:
Compute $mP$; compute $nQ$; add.

e.g. $b = 256$:
≈256 doublings for $mP$,
≈256 doublings for $nQ$,
≈50 additions for $mP$,
≈50 additions for $nQ$.

Worst case: $\approx b$ doublings plus
$\approx b/3$ additions of $\pm P$ or $\pm 3P$.
On average $\approx b/4$ additions.

Width-3 signed sliding windows:
Precompute $P, 3P, 5P, 7P$.
On average $\approx b/5$ additions.

On average $\approx b/6$ additions.

Cost of precomputation
eventually outweighs savings.
Optimal: $\approx b$ doublings plus
roughly $b/\log b$ additions.
Worst case: $\approx b$ doublings plus $\approx b/3$ additions of $\pm P$ or $\pm 3P$. On average $\approx b/4$ additions.

Width-3 signed sliding windows:
Precompute $P, 3P, 5P, 7P$. On average $\approx b/5$ additions.


Cost of precomputation eventually outweighs savings.
Optimal: $\approx b$ doublings plus roughly $b/\lg b$ additions.

Double-scalar multiplication
Want to quickly compute $m, P, n, Q \mapsto mP + nQ$.

e.g. verify signature $(R, S)$ by computing $h = H(R, M)$, computing $SB - hA$, checking whether $R = SB - hA$.

Obvious approach:
Compute $mP$; compute $nQ$; add.

e.g. $b = 256$:
$\approx 256$ doublings for $mP$, $\approx 256$ doublings for $nQ$, $\approx 50$ additions for $mP$, $\approx 50$ additions for $nQ$. 
Worst case: $\approx b$ doublings plus
additions of $\pm P$ or $\pm 3P$.
On average $\approx b/4$ additions.

Width-3 signed sliding windows:
Precompute $P, 3P, 5P, 7P$.
On average $\approx b/5$ additions.

Obvious approach:
On average $\approx b/6$ additions.

Cost of precomputation
eventually outweighs savings.

Optimal: $\approx b$ doublings plus
roughly $b = \lg b$ additions.

Double-scalar multiplication
Want to quickly compute $m, P, n, Q \mapsto mP + nQ$.
e.g. verify signature $(R, S)$
by computing $h = H(R, M)$,
computing $SB - hA$,
checking whether $R = SB - hA$.

Obvious approach:
Compute $mP$; compute $nQ$; add.
e.g. $b = 256$:
$\approx 256$ doublings for $mP$,
$\approx 256$ doublings for $nQ$,
$\approx 50$ additions for $mP$,
$\approx 50$ additions for $nQ$.

Joint doublings
Do much better by merging
$2X + 2Y$ into $2(X + Y)$.

def scalarmult2(m,P,n,Q):
if m == 0:
    return scalarmult(n,Q)
if n == 0:
    return scalarmult(m,P)
R = scalarmult2(m//2,P,n//2,Q)
R = R + R
if m % 2: R = R + P
if n % 2: R = R + Q
return R
Double-scalar multiplication

Want to quickly compute \(m, P, n, Q \mapsto mP + nQ\).

e.g. verify signature \((R, S)\) by computing \(h = H(R, M)\), computing \(SB - hA\), checking whether \(R = SB - hA\).

Obvious approach:
Compute \(mP\); compute \(nQ\); add.

\[\text{e.g. } b = 256:\]
\(\approx 256\) doublings for \(mP\),
\(\approx 256\) doublings for \(nQ\),
\(\approx 50\) additions for \(mP\),
\(\approx 50\) additions for \(nQ\).

Joint doublings

Do much better by merging \(2X + 2Y\) into \(2(X + Y)\).

\[
\begin{align*}
def \text{scalarmult2}(m, P, n, Q): \\
&\quad \text{if } m == 0: \\
&\quad \quad \text{return scalarmult}(n, Q) \\
&\quad \text{if } n == 0: \\
&\quad \quad \text{return scalarmult}(m, P) \\
&\quad R = \text{scalarmult2}(m//2, P, n//2, Q) \\
&\quad R = R + R \\
&\quad \text{if } m \% 2: R = R + P \\
&\quad \text{if } n \% 2: R = R + Q \\
&\quad \text{return } R
\end{align*}
\]
Double-scalar multiplication

Want to quickly compute \( m, P, n, Q \mapsto mP + nQ \).

e.g. verify signature \((R, S)\) by computing \( h = H(R, M)\), computing \( SB - hA\), checking whether \( R = SB - hA\).

Obvious approach: Compute \( mP \); compute \( nQ \); add.

e.g. \( b = 256\):
\( \approx 256 \) doublings for \( mP \),
\( \approx 256 \) doublings for \( nQ \),
\( \approx 50 \) additions for \( mP \),
\( \approx 50 \) additions for \( nQ \).

Joint doublings

Do much better by merging \( 2X + 2Y \) into \( 2(X + Y) \).

```python
def scalarmult2(m,P,n,Q):
    if m == 0:
        return scalarmult(n,Q)
    if n == 0:
        return scalarmult(m,P)
    R = scalarmult2(m//2,P,n//2,Q)
    R = R + R
    if m % 2: R = R + P
    if n % 2: R = R + Q
    return R
```
Double-scalar multiplication

Want to quickly compute \( m, P, n, Q \mapsto mP + nQ \).

E.g. verify signature \( (R, S) \) by computing \( h = H(R, M) \), computing \( SB - hA \), checking whether \( R = SB - hA \).

Obvious approach:
Compute \( mP \); compute \( nQ \); add.

E.g. \( b = 256 \):
\( \approx256 \) doublings for \( mP \),
\( \approx256 \) doublings for \( nQ \),
\( \approx50 \) additions for \( mP \),
\( \approx50 \) additions for \( nQ \).

Joint doublings

Do much better by merging \( 2X + 2Y \) into \( 2(X + Y) \).

```python
def scalarmult2(m,P,n,Q):
    if m == 0:
        return scalarmult(n,Q)
    if n == 0:
        return scalarmult(m,P)
    R = scalarmult2(m//2,P,n//2,Q)
    R = R + R
    if m % 2: R = R + P
    if n % 2: R = R + Q
    return R
```
Double-scalar multiplication
Want to quickly compute $m;P;n;Q \mapsto mP + nQ$.

Verify signature $(R, S)$ computing $h = H(R, M)$, computing $SB - hA$, checking whether $R = SB - hA$.

Obvious approach:
Compute $mP$; compute $nQ$; add.

E.g., $b = 256$:
$\approx 256$ doublings for $mP$,
$\approx 256$ doublings for $nQ$,
$\approx 50$ additions for $mP$,
$\approx 50$ additions for $nQ$.

Joint doublings
Do much better by merging $2X + 2Y$ into $2(X + Y)$.

```python
def scalarmult2(m, P, n, Q):
    if m == 0:
        return scalarmult(n, Q)
    if n == 0:
        return scalarmult(m, P)
    R = scalarmult2(m//2, P, n//2, Q)
    R = R + R
    if m % 2: R = R + P
    if n % 2: R = R + Q
    return R
```

For example: merge
$35P = 2(2(2(2(2P))) + P) + P$,
$31Q = 2(2(2(2Q + Q) + Q) + Q)$
into $35P + 31Q$...

Combine idea with windows:
E.g.,
$\approx b$ doublings (merged!),
$\approx b/2$ additions for $mP$,
$\approx b/2$ additions for $Q$.
$\approx 256$ doublings for $b = 256$,
$\approx 50$ additions using $P$,
$\approx 50$ additions using $Q$. 
Double-scalar multiplication

Want to quickly compute \( mP + nQ \).

e.g. verify signature \((R,S)\) by computing \( h = H(R,M) \), computing \( SB - hA \), checking whether \( R = SB - hA \).

Obvious approach: Compute \( mP \); compute \( nQ \); add.

e.g. \( b = 256 \):
\( \approx 256 \) doublings for \( mP \),
\( \approx 256 \) doublings for \( nQ \),
\( \approx 50 \) additions for \( mP \),
\( \approx 50 \) additions for \( nQ \).

Joint doublings

Do much better by merging \( 2X + 2Y \) into \( 2(X + Y) \).

```python
def scalarmult2(m, P, n, Q):
    if m == 0:
        return scalarmult(n, Q)
    if n == 0:
        return scalarmult(m, P)
    R = scalarmult2(m//2, P, n//2, Q)
    R = R + R
    if m % 2: R = R + P
    if n % 2: R = R + Q
    return R
```

For example: merge \( 35P = 2(2(2(2(2P ))) + P ) + P \), \( 31Q = 2(2(2(2Q + Q )+ Q )+ Q )+ Q \) into \( 35P + 31Q = 2(2(2(2(2P +Q )+Q )+Q )+Q ) + P+Q \).

\( \approx b \) doublings (merged!),
\( \approx b/2 \) additions of \( P \),
\( \approx b/2 \) additions of \( Q \).

Combine idea with windows: e.g.,
\( \approx 256 \) doublings for \( b = 256 \),
\( \approx 50 \) additions using \( P \),
\( \approx 50 \) additions using \( Q \).
Joint doublings

Do much better by merging $2X + 2Y$ into $2(X + Y)$.

```python
def scalarmult2(m,P,n,Q):
    if m == 0:
        return scalarmult(n,Q)
    if n == 0:
        return scalarmult(m,P)
    R = scalarmult2(m//2,P,n//2,Q)
    R = R + R
    if m % 2: R = R + P
    if n % 2: R = R + Q
    return R
```

For example: merge

$$35P = 2(2(2(2(2P))) + P) + P + P + P,$$
$$31Q = 2(2(2(2Q + Q) + Q) + Q) + Q + Q + Q + Q + Q$$

into $35P + 31Q = 2(2(2(2(2P + Q) + Q) + Q) + P + P + Q) + P + P + Q$.

$\approx b$ doublings (merged!),
$\approx b/2$ additions of $P$,
$\approx b/2$ additions of $Q$.

Combine idea with windows:

$\approx 256$ doublings for $b = 256$,
$\approx 50$ additions using $P$,
$\approx 50$ additions using $Q$. 
Joint doublings

Do much better by merging

\[ 2X + 2Y \]

into

\[ 2(X + Y) \].

def scalarmult2(m,P,n,Q):
    if m == 0:
        return scalarmult(n,Q)
    if n == 0:
        return scalarmult(m,P)
    R = scalarmult2(m//2,P,n//2,Q)
    R = R + R
    if m % 2: R = R + P
    if n % 2: R = R + Q
    return R

For example: merge

\[ 35P = 2(2(2(2P)) + P) + P, \]
\[ 31Q = 2(2(2Q+Q)+Q)+Q+Q \]

into

\[ 35P + 31Q = 2(2(2(2P+Q)+Q)+Q)+P+Q \]

\[ \approx b \text{ doublings (merged!)}, \]
\[ \approx b/2 \text{ additions of } P, \]
\[ \approx b/2 \text{ additions of } Q. \]

Combine idea with windows: e.g.,

\[ \approx 256 \text{ doublings for } b = 256, \]
\[ \approx 50 \text{ additions using } P, \]
\[ \approx 50 \text{ additions using } Q. \]
Joint doublings

Do much better by merging $2X + 2Y$ into $2(X + Y)$.

```python
def scalarmult2(m, P, n, Q):
    if m == 0:
        return scalarmult(n, Q)
    if n == 0:
        return scalarmult(m, P)
    R = scalarmult2(m // 2, P, n // 2, Q)
    R = R + R
    if m % 2: R = R + P
    if n % 2: R = R + Q
    return R
```

For example: merge

$$35P = 2(2(2(2P)) + P) + P,$$

$$31Q = 2(2(2Q+Q)+Q)+Q$$

into $35P + 31Q = 2(2(2(2P+Q)+Q)+Q)+P+Q$.

$\approx b$ doublings (merged!),

$\approx b/2$ additions of $P$,

$\approx b/2$ additions of $Q$.

Combine idea with windows: e.g.,

$\approx 256$ doublings for $b = 256$,

$\approx 50$ additions using $P$,

$\approx 50$ additions using $Q$.

Batch verification

Verifying many signatures:

need to be confident that

$S_1B = R_1 + h_1A_1$,

$S_2B = R_2 + h_2A_2$,

$S_3B = R_3 + h_3A_3$,

etc.

Obvious approach:

Check each equation separately.
Joint doublings
Do much better by merging $2X + 2Y$ into $2(X + Y)$.

```python
def scalarmult2(m,P,n,Q):
    if m == 0:
        return scalarmult(n,Q)
    if n == 0:
        return scalarmult(m,P)
    R = scalarmult2(m//2,P,n//2,Q)
    R = R + R
    if m % 2: R = R + P
    if n % 2: R = R + Q
    return R
```

For example: merge
$$35P = 2(2(2(2(2P)))+P)+P,$$
$$31Q = 2(2(2(2Q+Q)+Q)+Q)+Q$$
into $35P + 31Q =$$
$$2(2(2(2P+Q)+Q)+Q)+P+Q) + P+Q.$$

$\approx b$ doublings (merged!),
$\approx b/2$ additions of $P$,
$\approx b/2$ additions of $Q$.

Combine idea with windows: e.g.,
$\approx 256$ doublings for $b=256$,
$\approx 50$ additions using $P$,$\approx 50$ additions using $Q$.

Batch verification
Verifying many signatures:
need to be confident that
$S_1B = R_1 + h_1A_1$,
$S_2B = R_2 + h_2A_2$,
$S_3B = R_3 + h_3A_3$,
etc.

Obvious approach:
Check each equation separately.
Joint doublings
Do much better by merging $2^X + 2^Y$ into $2^{X+Y}$.

```python
def scalarmult2(m,P,n,Q):
    if m == 0:
        return scalarmult(n,Q)
    if n == 0:
        return scalarmult(m,P)
    R = scalarmult2(m//2,P,n//2,Q)
    R = R + R
    if m % 2: R = R + P
    if n % 2: R = R + Q
    return R
```

For example: merge $35P = 2(2(2(2P))) + P$, $31Q = 2(2(2Q+Q)+Q)+Q$ into $35P + 31Q = 2(2(2(2P+Q)+Q)+Q)+P+Q$.

$\approx b$ doublings (merged!),
$\approx b/2$ additions of $P$,
$\approx b/2$ additions of $Q$.

Combine idea with windows: e.g.,
$\approx 256$ doublings for $b = 256$,
$\approx 50$ additions using $P$,
$\approx 50$ additions using $Q$.

Batch verification
Verifying many signatures:
need to be confident that $S_1B = R_1 + h_1A_1$, $S_2B = R_2 + h_2A_2$, $S_3B = R_3 + h_3A_3$, etc.

Obvious approach:
Check each equation separately.
For example: merge

$35P = 2(2(2(2P))) + P + P,$
$31Q = 2(2(2Q+Q)+Q)+Q$ into $35P + 31Q =
2(2(2(2P+Q)+Q)+Q)+P+Q$.

$\approx b$ doublings (merged!),
$\approx b/2$ additions of $P$,
$\approx b/2$ additions of $Q$.

Combine idea with windows: e.g.,
$\approx 256$ doublings for $b = 256$,
$\approx 50$ additions using $P$,
$\approx 50$ additions using $Q$.

Batch verification

Verifying many signatures:

need to be confident that

$S_1 B = R_1 + h_1 A_1,$
$S_2 B = R_2 + h_2 A_2,$
$S_3 B = R_3 + h_3 A_3,$
etc.

Obvious approach:

Check each equation separately.
For example: merge
\[35P = 2(2(2(2(2P)))) + P,\]
\[31Q = 2(2(2(2Q+Q)+Q)+Q)+Q\]
into \[35P + 31Q = 2(2(2(2P+Q)+Q)+Q)+P+Q\].

\[\approx b\] doublings (merged!),
\[\approx b/2\] additions of \(P\),
\[\approx b/2\] additions of \(Q\).

Combine idea with windows: e.g.,
\[\approx 256 \text{ doublings for } b = 256,\]
\[\approx 50 \text{ additions using } P,\]
\[\approx 50 \text{ additions using } Q.\]

---

Batch verification

Verifying many signatures:
need to be confident that
\[S_1 B = R_1 + h_1 A_1,\]
\[S_2 B = R_2 + h_2 A_2,\]
\[S_3 B = R_3 + h_3 A_3,\]
etc.

Obvious approach:
Check each equation separately.

Much faster approach:
Check random linear combination of the equations.
Batch verification

Verifying many signatures: need to be confident that

\[ S_1 B = R_1 + h_1 A_1, \]
\[ S_2 B = R_2 + h_2 A_2, \]
\[ S_3 B = R_3 + h_3 A_3, \]

etc.

Obvious approach: Check each equation separately.

Much faster approach: Check random linear combination of the equations.

Easy to prove: forgeries have probability \( \leq 2^{-128} \) of fooling this check.
Batch verification

Verifying many signatures:
need to be confident that
\[ S_1 B = R_1 + h_1 A_1, \]
\[ S_2 B = R_2 + h_2 A_2, \]
\[ S_3 B = R_3 + h_3 A_3, \]

etc.

Obvious approach:
Check each equation separately.

Much faster approach:
Check random linear combination of the equations.

Easy to prove:
forgeries have probability \( \leq 2^{-128} \) of fooling this check.

Pick independent uniform random 128-bit \( z_1, z_2, z_3, \ldots \).
Check whether
\[
(z_1 S_1 + z_2 S_2 + z_3 S_3 + \cdots) B = z_1 R_1 + (z_1 h_1) A_1 + z_2 R_2 + (z_2 h_2) A_2 + z_3 R_3 + (z_3 h_3) A_3 + \cdots.
\]

(If \( \neq \): See 2012 Bernstein–Doumen–Lange–Oosterwijk.)
Batch verification

Verifying many signatures: need to be confident that

\[ S_1 B = R_1 + h_1 A_1, \]
\[ S_2 B = R_2 + h_2 A_2, \]
\[ S_3 B = R_3 + h_3 A_3, \]
etc.

Obvious approach: Check each equation separately.

Much faster approach: Check random linear combination of the equations.

(If \( \neq \): See 2012 Bernstein–Doumen–Lange–Oosterwijk.)

Easy to prove: forgeries have probability \( \leq 2^{-128} \) of fooling this check.

Pick independent uniform random 128-bit \( z_1, z_2, z_3, \ldots \).

Check whether

\[ (z_1 S_1 + z_2 S_2 + z_3 S_3 + \cdots) B = z_1 R_1 + (z_1 h_1) A_1 + \]
\[ z_2 R_2 + (z_2 h_2) A_2 + \]
\[ z_3 R_3 + (z_3 h_3) A_3 + \cdots. \]
Batch verification

Verifying many signatures:
need to be confident that
\[ S_1 B = R_1 + h_1 A_1, \]
\[ S_2 B = R_2 + h_2 A_2, \]
\[ S_3 B = R_3 + h_3 A_3, \]

etc.

Obvious approach:
Check each equation separately.

Much faster approach:
Check random linear combination of the equations.

Pick independent uniform random 128-bit \( z_1, z_2, z_3, \ldots \).

Check whether
\[ (z_1 S_1 + z_2 S_2 + z_3 S_3 + \cdots) B = z_1 R_1 + (z_1 h_1) A_1 + z_2 R_2 + (z_2 h_2) A_2 + z_3 R_3 + (z_3 h_3) A_3 + \cdots. \]

(If \( \neq \): See 2012 Bernstein–Doumen–Lange–Oosterwijk.)

Easy to prove:
forgeries have probability \( \leq 2^{-128} \) of fooling this check.
Batch verification

Verifying many signatures:
need to be confident that
\[ S_1 B = R_1 + h_1 A_1, \]
\[ S_2 B = R_2 + h_2 A_2, \]
\[ S_3 B = R_3 + h_3 A_3, \]
\[ \text{etc.} \]

Obvious approach:
Check each equation separately.

Much faster approach:
Check random linear combination
of the equations.

Pick independent uniform random 128-bit \( z_1, z_2, z_3, \ldots \).

Check whether
\[ (z_1 S_1 + z_2 S_2 + z_3 S_3 + \cdots) B = \]
\[ z_1 R_1 + (z_1 h_1) A_1 + \]
\[ z_2 R_2 + (z_2 h_2) A_2 + \]
\[ z_3 R_3 + (z_3 h_3) A_3 + \cdots. \]

(If \( \neq \): See 2012 Bernstein–Doumen–Lange–Oosterwijk.)

Easy to prove:
forgeries have probability \( \leq 2^{-128} \)
of fooling this check.

Multi-scalar multiplication

Review of asymptotic speeds:
1939 Brauer (windows):
\[ \approx (1 + 1 \div \log b) b \]
additions to compute \( P \mapsto nP \) if \( n < 2^b \).

1964 Straus (joint doublings):
\[ \approx (1 + k \div \log b) b \]
additions to compute \( P_1, \ldots, P_k \mapsto n_1 P_1 + \cdots + n_k P_k \) if \( n_1, \ldots, n_k < 2^b \).
Batch verification

Verifying many signatures:
need to be confident that
$S_1 B = R_1 + h_1 A_1,$
$S_2 B = R_2 + h_2 A_2,$
$S_3 B = R_3 + h_3 A_3,$
etc.

Obvious approach:
Check each equation separately.

Much faster approach:
Check random linear combination
of the equations.

Pick independent uniform random 128-bit $z_1, z_2, z_3, \ldots$.

Check whether
$(z_1 S_1 + z_2 S_2 + z_3 S_3 + \cdots) B = z_1 R_1 + (z_1 h_1) A_1 +$
$z_2 R_2 + (z_2 h_2) A_2 +$
$z_3 R_3 + (z_3 h_3) A_3 + \cdots.$

(If $\neq$: See 2012 Bernstein–Doumen–Lange–Oosterwijk.)

Easy to prove:
forgeries have probability $\leq 2^{-128}$ of fooling this check.

Multi-scalar multiplication

Review of asymptotic speeds:

1939 Brauer (windows):
$\approx (1 + 1/\lg b) b$
additions to compute $P \mapsto n P$ if $n < 2^b$.

1964 Straus (joint doublings):
$\approx (1 + k/\lg b) b$
additions to compute $P_1, \ldots, P_k \mapsto n_1 P_1 + \cdots + n_k P_k$ if $n_1, \ldots, n_k < 2^b$. 

Batch verification

Verifying many signatures:

need to be confident that

\[ S_1 B = R_1 + h_1 A_1, \]
\[ S_2 B = R_2 + h_2 A_2, \]
\[ S_3 B = R_3 + h_3 A_3, \]

etc.

Obvious approach:

Check each equation separately.

Much faster approach:

Check random linear combination

of the equations.

---

Pick independent uniform random

128-bit \( z_1, z_2, z_3, \ldots \). 

Check whether 

\[(z_1 S_1 + z_2 S_2 + z_3 S_3 + \cdots) B = z_1 R_1 + (z_1 h_1) A_1 + z_2 R_2 + (z_2 h_2) A_2 + z_3 R_3 + (z_3 h_3) A_3 + \cdots.\]

(If \( \neq \)

See 2012 Bernstein–Doumen–Lange–Oosterwijk.)

Easy to prove:

forgeries have probability \( \leq 2^{-128} \) of fooling this check.

---

Multi-scalar multiplication

Review of asymptotic speeds:

1939 Brauer (windows):

\[ \approx (1 + 1/\log b) b \]

additions to compute 

\[ P \mapsto n P \text{ if } n < 2^b. \]

1964 Straus (joint doublings):

\[ \approx (1 + k/\log b) b \]

additions to compute 

\[ P_1, \ldots, P_k \mapsto n_1 P_1 + \cdots + n_k P_k \text{ if } n_1, \ldots, n_k < 2^b. \]
Pick independent uniform random 128-bit $z_1, z_2, z_3, \ldots$.

Check whether
\[(z_1 S_1 + z_2 S_2 + z_3 S_3 + \cdots) B = z_1 R_1 + (z_1 h_1) A_1 +
\]
\[z_2 R_2 + (z_2 h_2) A_2 +
\]
\[z_3 R_3 + (z_3 h_3) A_3 + \cdots.
\]

(If $\neq$: See 2012 Bernstein–Doumen–Lange–Oosterwijk.)

Easy to prove:
forgeries have probability $\leq 2^{-128}$ of fooling this check.

Multi-scalar multiplication
Review of asymptotic speeds:

1939 Brauer (windows):
\[\approx (1 + 1/\lg b) b\]
additions to compute $P \mapsto nP$ if $n < 2^b$.

1964 Straus (joint doublings):
\[\approx (1 + k/\lg b) b\]
additions to compute $P_1, \ldots, P_k \mapsto n_1 P_1 + \cdots + n_k P_k$ if $n_1, \ldots, n_k < 2^b$. 
Pick independent uniform random \( z_1, z_2, z_3, \ldots \).

Check whether
\[
(z_1 S_1 + z_2 S_2 + z_3 S_3 + \cdots) = B = (z_1 h_1) A_1 + (z_2 h_2) A_2 + (z_3 h_3) A_3 + \cdots.
\]

See 2012 Bernstein–Doumen–Lange–Oosterwijk.)

Easy to prove:
forgeries have probability \( \leq 2^{-128} \) of fooling this check.

Multi-scalar multiplication

Review of asymptotic speeds:

1939 Brauer (windows):
\[
\approx (1 + 1 = \lg b) b
\]
additions to compute \( P \mapsto n P \) if \( n < 2^b \).

1964 Straus (joint doublings):
\[
\approx (1 + k = \lg b) b
\]
additions to compute \( P_1, \ldots, P_k \mapsto n_1 P_1 + \cdots + n_k P_k \) if \( n_1, \ldots, n_k < 2^b \).

1976 Yao:
\[
\approx (1 + k = \lg b) b
\]
additions to compute \( P \mapsto n P \) if \( n_1, \ldots, n_k < 2^b \).

1976 Pippenger:
Similar asymptotics, but replace \( \lg b \) with \( \lg(kb) \).
Faster than Straus and Yao if \( k \) is large.
(Knuth says "generalization" as if speed were the same.)
Multi-scalar multiplication

Review of asymptotic speeds:

1939 Brauer (windows):
\[ \approx (1 + 1/\lg b) b \]
additions to compute
\[ P \mapsto nP \text{ if } n < 2^b. \]

1964 Straus (joint doublings):
\[ \approx (1 + k/\lg b) b \]
additions to compute
\[ P_1, \ldots, P_k \mapsto n_1 P_1 + \cdots + n_k P_k \]
if \( n_1, \ldots, n_k < 2^b \).

1976 Yao:
\[ \approx (1 + k/\lg b) b \]
additions to compute
\[ P \mapsto n_1 P, \ldots, n_k P \]
if \( n_1, \ldots, n_k < 2^b \).

1976 Pippenger:
Similar asymptotics, but replace \( \lg b \) with \( \lg(k b) \).
Faster than Straus and Yao if \( k \) is large.
(Knuth says “generalization” as if speed were the same.)
Multi-scalar multiplication

Review of asymptotic speeds:

1939 Brauer (windows):

\[ \approx (1 + 1/\lg b) b \]

additions to compute

\[ P \mapsto nP \text{ if } n < 2^b. \]

1964 Straus (joint doublings):

\[ \approx (1 + k/\lg b) b \]

additions to compute

\[ P_1, \ldots, P_k \mapsto n_1 P_1 + \cdots + n_k P_k \]

if \( n_1, \ldots, n_k < 2^b. \)

1976 Yao:

\[ \approx (1 + k/\lg b) b \]

additions to compute

\[ P \mapsto n_1 P, \ldots, n_k P \]

if \( n_1, \ldots, n_k < 2^b. \)

1976 Pippenger:

Similar asymptotics, but replace \( \lg b \) with \( \lg(kb) \).

Faster than Straus and Yao if \( k \) is large.

(Knuth says “generalization” as if speed were the same.)
Multi-scalar multiplication

Review of asymptotic speeds:

1939 Brauer (windows):
\[ \approx (1 + 1/\lg b) b \]
additions to compute
\( P \mapsto nP \) if \( n < 2^b \).

1964 Straus (joint doublings):
\[ \approx (1 + k/\lg b) b \]
additions to compute
\( P_1, \ldots, P_k \mapsto n_1 P_1 + \cdots + n_k P_k \)
if \( n_1, \ldots, n_k < 2^b \).

1976 Yao:
\[ \approx (1 + k/\lg b) b \]
additions to compute
\( P \mapsto n_1 P, \ldots, n_k P \)
if \( n_1, \ldots, n_k < 2^b \).

1976 Pippenger:
Similar asymptotics,
but replace \( \lg b \) with \( \lg(kb) \).
Faster than Straus and Yao
if \( k \) is large.
(Knuth says “generalization”
as if speed were the same.)
Multi-scalar multiplication

Review of asymptotic speeds:

1939 Brauer (windows):
\[ \approx (1 + \frac{1}{\log b}) b \]
additions to compute \( P \mapsto n_P \) if \( n < 2^b \).

1964 Straus (joint doublings):
\[ \approx (1 + \frac{k}{\log b}) b \]
additions to compute \( P_1 \mapsto n_1 P_1 + \cdots + n_k P_k \) if \( n_1, \ldots, n_k < 2^b \).

1976 Yao:
\[ \approx (1 + \frac{k}{\log b}) b \]
additions to compute \( P \mapsto n_1 P_1 + \cdots + n_k P_k \) if \( n_1, \ldots, n_k < 2^b \).

1976 Pippenger:
Similar asymptotics, but replace \( \log b \) with \( \log(kb) \).
Faster than Straus and Yao if \( k \) is large.
(Knuth says “generalization” as if speed were the same.)

More generally, Pippenger’s algorithm computes \( \ell \) sums of multiples of \( k \) inputs.
\[ \approx \left( \min \left\{ \frac{k}{\log \left( \frac{k}{b} \right)} \right\} + \frac{k}{\log(kb)} \right) b \]
\(< b \) adds if all coefficients are below \( 2^b \).
Within \( 1 + \varepsilon \) of optimal.
Multi-scalar multiplication

Review of asymptotic speeds:

1939 Brauer (windows):
\[ \approx (1 + \frac{1}{\log b}) \cdot b \]
additions to compute \( P \mapsto n P \) if \( n < 2^b \).

1964 Straus (joint doublings):
\[ \approx (1 + \frac{k}{\log b}) \cdot b \]
additions to compute \( P_1 \mapsto n_1 P_1 + \cdots + n_k P_k \) if \( n_1, \ldots, n_k < 2^b \).

1976 Yao:
\[ \approx (1 + \frac{k}{\log b}) \cdot b \]
additions to compute \( P_1 \mapsto n_1 P_1 + \cdots + n_k P_k \) if \( n_1, \ldots, n_k < 2^b \).

1976 Pippenger:
Similar asymptotics, but replace \( \log b \) with \( \log(k^*b) \).
Faster than Straus and Yao if \( k \) is large.
(Knuth says "generalization" as if speed were the same.)

More generally, Pippenger's algorithm computes \( \ell \) sums of multiples of \( k \) inputs.
\[ \approx \left( \min \{k, \ell\} + \frac{\ell}{\log(k^*b)} \right) \cdot b \]
if all coefficients are below \( 2^b \).
Within \( 1 + \epsilon \) of optimal.
21
1939 Brauer (windows):
\[ \approx (1 + \frac{1}{\lg b}) b \]
additions to compute \( P \mapsto n P \) if \( n < 2^b \).

1964 Straus (joint doublings):
\[ \approx (1 + \frac{k}{\lg b}) b \]
additions to compute \( P_1, \ldots, P_k \mapsto n_1 P_1 + \cdots + n_k P_k \) if \( n_1, \ldots, n_k < 2^b \).

1976 Yao:
\[ \approx (1 + \frac{k}{\lg b}) b \]
additions to compute \( P \mapsto n_1 P; \ldots; n_k P \) if \( n_1, \ldots, n_k < 2^b \).

1976 Pippenger:
Similar asymptotics, but replace \( \lg b \) with \( \lg(kb) \).
Faster than Straus and Yao if \( k \) is large.
(Knuth says “generalization” as if speed were the same.)

22
More generally, Pippenger’s algorithm computes \( \ell \) sums of multiples of \( k \) inputs:
\[ \approx \left( \min\{k, \ell\} + \frac{k\ell}{\lg(kb)} \right) b \]
if all coefficients are below \( 2^b \).
Within \( 1 + \epsilon \) of optimal.
1976 Yao:
≈ (1 + \(k/\lg b\))b

additions to compute
\(P \mapsto n_1 P, \ldots, n_k P\)
if \(n_1, \ldots, n_k < 2^b\).

1976 Pippenger:

Similar asymptotics,
but replace \(\lg b\) with \(\lg(kb)\).
Faster than Straus and Yao
if \(k\) is large.

(Knuth says “generalization”
as if speed were the same.)

More generally, Pippenger’s algorithm computes
\(\ell\) sums of multiples of \(k\) inputs.

\[\approx \left( \min\{k, \ell\} + \frac{k\ell}{\lg(k\ell b)} \right) b \text{ adds}\]
if all coefficients are below \(2^b\).
Within \(1 + \epsilon\) of optimal.
1976 Yao:

\[ \approx (1 + k/\lg b)b \]

additions to compute

\[ P \mapsto n_1 P, \ldots, n_k P \]

if \( n_1, \ldots, n_k < 2^b \).

1976 Pippenger:

Similar asymptotics,
but replace \( \lg b \) with \( \lg(kb) \).
Faster than Straus and Yao
if \( k \) is large.

(Knuth says “generalization”
as if speed were the same.)

More generally, Pippenger’s algorithm computes

\[ \ell \text{ sums of multiples of } k \text{ inputs.} \]

\[ \approx \left( \min\{k, \ell\} + \frac{k\ell}{\lg(k\ell b)} \right) b \text{ adds} \]

if all coefficients are below \( 2^b \).
Within \( 1 + \epsilon \) of optimal.

Various special cases of
Pippenger’s algorithm were reinvented and patented by
Is that the end of the story?
More generally, Pippenger’s algorithm computes \( \ell \) sums of multiples of \( k \) inputs. \( \approx \left( \min\{k, \ell\} + \frac{k\ell}{\log(kb)} \right) b \) adds if all coefficients are below \( 2^b \). Within \( 1 + \epsilon \) of optimal.

Various special cases of Pippenger’s algorithm were reinvented and patented by 1993 Brickell–Gordon–McCurley–Wilson, 1995 Lim–Lee, etc. Is that the end of the story?
More generally, Pippenger’s algorithm computes $\ell$ sums of multiples of $k$ inputs.

$$\approx \left( \min\{k, \ell\} + \frac{kl}{\lg(klb)} \right) b$$

adds if all coefficients are below $2^b$. Within $1 + \epsilon$ of optimal.

Various special cases of Pippenger’s algorithm were reinvented and patented by 1993 Brickell–Gordon–McCurley–Wilson, 1995 Lim–Lee, etc.

Is that the end of the story?
22

More generally, Pippenger’s algorithm computes
\( \ell \) sums of multiples of \( k \) inputs.

\[ \approx \left( \min\{k, \ell\} + \frac{kl}{\lg(klb)} \right) b \text{ adds} \]

if all coefficients are below \( 2^b \).

Within \( 1 + \epsilon \) of optimal.

Various special cases of Pippenger’s algorithm were reinvented and patented by 1993 Brickell–Gordon–McCurley–Wilson, 1995 Lim–Lee, etc.

Is that the end of the story?

23

No! 1989 Bos–Coster:

If \( n_1 \geq n_2 \geq \cdots \) then

\[ n_1 P_1 + n_2 P_2 + n_3 P_3 + \cdots = \]

\[ (n_1 - qn_2)P_1 + n_2(qP_1 + P_2) + n_3 P_3 + \cdots \text{ where } q = \lfloor n_1/n_2 \rfloor. \]

Remarkably simple; competitive with Pippenger for random choices of \( n_i \)'s; much better memory usage.
More generally, Pippenger's algorithm computes \( \ell \) sums of multiples of \( k \) inputs.

\[
\approx \left( \min\{k, \ell\} + \frac{k \ell}{\lg(k \ell b)} \right) b \text{ adds if all coefficients are below } 2^b.
\]

Within \( 1 + \epsilon \) of optimal.

Various special cases of Pippenger's algorithm were reinvented and patented by 1993 Brickell–Gordon–McCurley–Wilson, 1995 Lim–Lee, etc.

Is that the end of the story?

No! 1989 Bos–Coster:

If \( n_1 \geq n_2 \geq \cdots \) then

\[
n_1 P_1 + n_2 P_2 + n_3 P_3 + \cdots = (n_1 - q n_2) P_1 + n_2 (q P_1 + P_2) + n_3 P_3 + \cdots \text{ where } q = \lfloor n_1/n_2 \rfloor.
\]

Remarkably simple; competitive with Pippenger for random choices of \( n_i \)'s; much better memory usage.
More generally, Pippenger’s algorithm computes sums of multiples of $k$ inputs.

$$\min\{k, l\} + \frac{kl}{\lg(klb)}$$

$b$ adds $b$ terms if all coefficients are below $2^b$.

$1 + \epsilon$ of optimal.

Various special cases of Pippenger’s algorithm were reinvented and patented by 1993 Brickell–Gordon–McCurley–Wilson, 1995 Lim–Lee, etc. Is that the end of the story?

No! 1989 Bos–Coster:

If $n_1 \geq n_2 \geq \cdots$ then

$$n_1P_1 + n_2P_2 + n_3P_3 + \cdots = (n_1 - qn_2)P_1 + n_2(qP_1 + P_2) + n_3P_3 + \cdots$$

where $q = \lfloor n_1/n_2 \rfloor$.

Remarkably simple; competitive with Pippenger for random choices of $n_i$’s; much better memory usage.

Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1

More generally, Pippenger’s algorithm computes sums of multiples of $k$ inputs.

$$\approx \min\{k;\} + k' \lg(k'b)$$

adds if all coefficients are below $2^b$. Within 1 + $\varepsilon$ of optimal.

Various special cases of Pippenger’s algorithm were reinvented and patented by 1993 Brickell–Gordon–McCurley–Wilson, 1995 Lim–Lee, etc. Is that the end of the story?

No! 1989 Bos–Coster:

If $n_1 \geq n_2 \geq \cdots$ then

$$n_1 P_1 + n_2 P_2 + n_3 P_3 + \cdots = (n_1 - qn_2)P_1 + n_2(qP_1 + P_2) + n_3P_3 + \cdots$$

where $q = \lfloor n_1/n_2 \rfloor$.

Remarkably simple; competitive with Pippenger for random choices of $n_i$’s; much better memory usage.

Example of Bos–Coster:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>000100000</td>
<td>32</td>
</tr>
<tr>
<td>000010000</td>
<td>16</td>
</tr>
<tr>
<td>100101100</td>
<td>300</td>
</tr>
<tr>
<td>010010010</td>
<td>146</td>
</tr>
<tr>
<td>001001101</td>
<td>77</td>
</tr>
<tr>
<td>000000010</td>
<td>2</td>
</tr>
<tr>
<td>000000001</td>
<td>1</td>
</tr>
</tbody>
</table>

More generally, Pippenger's algorithm computes sums of multiples of $k \text{ inputs}$. 

\[
\approx \min \{ k, \cdot \} + k' \lg (k'b) \\
\text{adds if all coefficients are below } 2^b.
\]

Within $1 + \epsilon$ of optimal.

Various special cases of Pippenger's algorithm were reinvented and patented by 1993 Brickell–Gordon–McCurley–Wilson, 1995 Lim–Lee, etc. Is that the end of the story?

No! 1989 Bos–Coster:

If $n_1 \geq n_2 \geq \cdots$ then 

\[
\sum n_i P_i = (n_1 - qn_2)P_1 + n_2(qP_1 + P_2) + n_3P_3 + \cdots \text{ where } q = \lfloor n_1/n_2 \rfloor.
\]

Remarkably simple; competitive with Pippenger for random choices of $n_i$'s; much better memory usage.

Example of Bos–Coster:

\[
\begin{align*}
000100000 &= 32 \\
000010000 &= 16 \\
100101100 &= 300 \\
010010010 &= 146 \\
001001101 &= 77 \\
000000010 &= 2 \\
000000001 &= 1
\end{align*}
\]


No! 1989 Bos–Coster:

If \( n_1 \geq n_2 \geq \cdots \) then

\[
\begin{align*}
  n_1 P_1 + n_2 P_2 + n_3 P_3 + \cdots &= \\
  (n_1 - qn_2)P_1 + n_2(qP_1 + P_2) + \\
  n_3 P_3 + \cdots \quad \text{where } q = \lfloor n_1/n_2 \rfloor.
\end{align*}
\]

Remarkably simple; competitive with Pippenger for random choices of \( n_i \)'s; much better memory usage.

Example of Bos–Coster:

\[
\begin{align*}
  000100000 &= 32 \\
  000010000 &= 16 \\
  100101100 &= 300 \\
  010010010 &= 146 \\
  001001101 &= 77 \\
  000000010 &= 2 \\
  000000001 &= 1
\end{align*}
\]

If $n_1 \geq n_2 \geq \cdots$ then

$$n_1 P_1 + n_2 P_2 + n_3 P_3 + \cdots = (n_1 - qn_2) P_1 + n_2 (qP_1 + P_2) + \cdots$$

where $q = \lfloor n_1 / n_2 \rfloor$.

Remarkably simple; competitive with Pippenger for random choices of $n_i$'s; much better memory usage.

Example of Bos–Coster:

<table>
<thead>
<tr>
<th>Number</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>000100000</td>
</tr>
<tr>
<td>16</td>
<td>000010000</td>
</tr>
<tr>
<td>300</td>
<td>100101100</td>
</tr>
<tr>
<td>146</td>
<td>010010010</td>
</tr>
<tr>
<td>77</td>
<td>001001101</td>
</tr>
<tr>
<td>2</td>
<td>000000010</td>
</tr>
<tr>
<td>1</td>
<td>000000001</td>
</tr>
</tbody>
</table>


Reduce largest row:

<table>
<thead>
<tr>
<th>Number</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>000100000</td>
</tr>
<tr>
<td>16</td>
<td>000010000</td>
</tr>
<tr>
<td>154</td>
<td>010011010</td>
</tr>
<tr>
<td>146</td>
<td>010010010</td>
</tr>
<tr>
<td>77</td>
<td>001001101</td>
</tr>
<tr>
<td>2</td>
<td>000000010</td>
</tr>
<tr>
<td>1</td>
<td>000000001</td>
</tr>
</tbody>
</table>

Plus one extra addition: add $146P$ into $154P$, obtaining $300P$.
Example of Bos–Coster:

\[\begin{align*}
000100000 &= 32 \\
000010000 &= 16 \\
100101100 &= 300 \\
010010010 &= 146 \\
001001101 &= 77 \\
000000010 &= 2 \\
000000001 &= 1
\end{align*}\]

Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1


Reduce largest row:

000100000 = 32
000010000 = 16
010011010 = 154 ←
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1


Plus one extra addition:

add $146P$ into $154P$, obtaining $300P$. 

Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1


Reduce largest row:

000100000 = 32
000010000 = 16
010011010 = 154 ←
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1


Plus one extra addition:
add $146_P$ into $154_P$, obtaining $300_P$. 
Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1


Reduce largest row:

000100000 = 32
000010000 = 16
000001000 = 8 ←
010010010 = 146
001001101 = 77
001001101 = 77
000000010 = 2
000000001 = 1

plus 2 additions.
Example of Bos–Coster:

\begin{align*}
000100000 &= 32 \\
000010000 &= 16 \\
100101100 &= 300 \\
010010010 &= 146 \\
001001101 &= 77 \\
000000010 &= 2 \\
000000001 &= 1
\end{align*}


Reduce largest row:

\begin{align*}
000100000 &= 32 \\
000010000 &= 16 \\
000001000 &= 8 \\
001000101 &= 69 \leftarrow \\
001001101 &= 77 \\
000000010 &= 2 \\
000000001 &= 1
\end{align*}

plus 3 additions.
Example of Bos–Coster:

- 000100000 = 32
- 000010000 = 16
- 100101100 = 300
- 010010010 = 146
- 001001101 = 77
- 000000010 = 2
- 000000001 = 1


Reduce largest row:

- 000100000 = 32
- 000010000 = 16
- 000001000 = 8
- 001000101 = 69
- 000001000 = 8 ←
- 000000010 = 2
- 000000001 = 1

plus 4 additions.
Example of Bos–Coster:

\[ \begin{align*}
000100000 &= 32 \\
000010000 &= 16 \\
100101100 &= 300 \\
010010010 &= 146 \\
001001101 &= 77 \\
000000010 &= 2 \\
000000001 &= 1
\end{align*} \]


Reduce largest row:

\[ \begin{align*}
000100000 &= 32 \\
000010000 &= 16 \\
000001000 &= 8 \\
000100101 &= 37 \\
000000101 &= 1
\end{align*} \]

plus 5 additions.
Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1


Reduce largest row:

000100000 = 32
000010000 = 16
000001000 = 8
000000101 = 5 ←
000001000 = 8
000000010 = 2
000000001 = 1

plus 6 additions.
Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1


Reduce largest row:

000010000 = 16 ←
000010000 = 16
000001000 = 8
000000101 = 5
000001000 = 8
000000010 = 2
000000001 = 1

plus 7 additions.
Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1


Reduce largest row:

000000000 = 0
000010000 = 16
000001000 = 8
000000101 = 5
000001000 = 8
000000010 = 2
000000001 = 1

plus 7 additions.
Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1

Goal: Compute 32\(P\), 16\(P\), 300\(P\), 146\(P\), 77\(P\), 2\(P\), 1\(P\).

Reduce largest row:

000000000 = 0
000001000 = 8 ←
000000101 = 5
000001000 = 8
000000010 = 2
000000001 = 1

plus 8 additions.
Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1


Reduce largest row:

000000000 = 0
000000000 = 0 ←
000001000 = 8
000000101 = 5
000001000 = 8
000000010 = 2
000000001 = 1

plus 8 additions.
Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1


Reduce largest row:

000000000 = 0
000000000 = 0
000000000 = 0 ←
000000101 = 5
000001000 = 8
000000010 = 2
000000001 = 1

plus 8 additions.
Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1


Reduce largest row:

000000000 = 0
000000000 = 0
000000000 = 0
000000101 = 5
000000011 = 3 ←
000000010 = 2
000000001 = 1

plus 9 additions.
Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1


Reduce largest row:

000000000 = 0
000000000 = 0
000000000 = 0
000000010 = 2 ←
000000011 = 3
000000010 = 2
000000001 = 1

plus 10 additions.
Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1

Goal: Compute \(32P\), \(16P\), \(300P\), \(146P\), \(77P\), \(2P\), \(1P\).

Reduce largest row:

000000000 = 0
000000000 = 0
000000000 = 0
000000010 = 2
000000001 = 1 ←
000000010 = 2
000000001 = 1

plus 11 additions.
Example of Bos–Coster:

$000100000 = 32$
$000010000 = 16$
$100101100 = 300$
$010010010 = 146$
$001001101 = 77$
$000000010 = 2$
$000000001 = 1$


Reduce largest row:

$000000000 = 0$
$000000000 = 0$
$000000000 = 0$
$000000000 = 0 \leftarrow$
$000000001 = 1$
$000000010 = 2$
$000000001 = 1$

plus 11 additions.
Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1


Reduce largest row:

000000000 = 0
000000000 = 0
000000000 = 0
000000000 = 0
000000001 = 1
000000001 = 1 ←
000000001 = 1
000000001 = 1

plus 12 additions.
Example of Bos–Coster:

- \(000100000 = 32\)
- \(000010000 = 16\)
- \(100101100 = 300\)
- \(010010010 = 146\)
- \(001001101 = 77\)
- \(000000010 = 2\)
- \(000000001 = 1\)

Goal: Compute \(32P\), \(16P\), \(300P\), \(146P\), \(77P\), \(2P\), \(1P\).

Reduce largest row:

- \(000000000 = 0\)
- \(000000000 = 0\)
- \(000000000 = 0\)
- \(000000000 = 0\)
- \(000000000 = 0\) ←
- \(000000001 = 1\)
- \(000000001 = 1\)

plus 12 additions.
Example of Bos–Coster:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>000100000</td>
<td>32</td>
</tr>
<tr>
<td>000010000</td>
<td>16</td>
</tr>
<tr>
<td>100101100</td>
<td>300</td>
</tr>
<tr>
<td>010010010</td>
<td>146</td>
</tr>
<tr>
<td>001001101</td>
<td>77</td>
</tr>
<tr>
<td>000000010</td>
<td>2</td>
</tr>
<tr>
<td>000000001</td>
<td>1</td>
</tr>
</tbody>
</table>


Reduce largest row:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000</td>
<td>0</td>
</tr>
<tr>
<td>000000001</td>
<td>1</td>
</tr>
</tbody>
</table>

plus 12 additions.
Example of Bos–Coster:

000100000 = 32
000010000 = 16
100101100 = 300
010010010 = 146
001001101 = 77
000000010 = 2
000000001 = 1


Reduce largest row:

000000000 = 0
000000000 = 0
000000000 = 0
000000000 = 0
000000000 = 0
000000000 = 0

← plus 12 additions.

Final addition chain: 1, 2, 3, 5, 8, 16, 32, 37, 69, 77, 146, 154, 300.

Short, no temporary storage, low two-operand complexity.
Example of Bos–Coster:

- \(000100000 = 32\)
- \(000010000 = 16\)
- \(100101100 = 300\)
- \(010010010 = 146\)
- \(001001101 = 77\)
- \(000000010 = 2\)
- \(000000001 = 1\)

Goal: Compute \(32P\), \(16P\), \(300P\), \(146P\), \(77P\), \(2P\), \(1P\).

Reduce largest row:

- \(000000000 = 0\)
- \(000000000 = 0\)
- \(000000000 = 0\)
- \(000000000 = 0\)
- \(000000000 = 0\)
- \(000000000 = 0\)
- \(000000000 = 0\)

Plus 12 additions.

Final addition chain: 1, 2, 3, 5, 8, 16, 32, 37, 69, 77, 146, 154, 300.

Short, no temporary storage, low two-operand complexity.

Revised goal: Compute

- \(32P_1 + 16P_2 + 300P_3 + 146P_4 + 77P_5 + 2P_6 + 1P_7\).

First compute \(P_4' = P_4 + P_3\) and then recursively compute

- \(32P_1 + 16P_2 + 154P_3 + 146P_4' + 77P_5 + 2P_6 + 1P_7\).

Same scalars show up as before.

Ed25519 batch verification:

Verify batch of 64 signatures about twice as fast as verifying each separately.
Example of Bos–Coster:

\begin{align*}
000100000 &= 32 \\
000010000 &= 16 \\
100101100 &= 300 \\
010010010 &= 146 \\
001001101 &= 77 \\
000000010 &= 2 \\
000000001 &= 1 \\
\end{align*}

Goal: Compute $32 P_1$, $16 P_2$, $300 P_3$, $146 P_4$, $77 P_5$, $2 P_6$, $1 P_7$.

Reduce largest row:

\begin{align*}
000000000 &= 0 \\
000000000 &= 0 \\
000000000 &= 0 \\
000000000 &= 0 \\
000000000 &= 0 \\
000000000 &= 0 \\
\end{align*}

plus 12 additions.

Final addition chain: 1, 2, 3, 5, 8, 16, 32, 37, 69, 77, 146, 154, 300.

Short, no temporary storage, low two-operand complexity.

Revised goal: Compute $32P_1 + 16P_2 + 300P_3 + 146P_4 + 77P_5 + 2P_6 + 1P_7$.

First compute $P'_4 = P_4 + P_3$ and then recursively compute

$32P_1 + 16P_2 + 154P_3 + 146P_4' + 77P_5 + 2P_6 + 1P_7$.

Same scalars show up as before.

Ed25519 batch verification:
verify batch of 64 signatures about twice as fast as verifying each separately.
Reduce largest row:

\[ 000000000 = 0 \]
\[ 000000000 = 0 \]
\[ 000000000 = 0 \]
\[ 000000000 = 0 \]
\[ 000000000 = 0 \]
\[ 000000000 = 0 \leftarrow \]

plus 12 additions.

Final addition chain: 1, 2, 3, 5, 8, 16, 32, 37, 69, 77, 146, 154, 300.

Short, no temporary storage, low two-operand complexity.

Revised goal: Compute
\[ 32P_1 + 16P_2 + 300P_3 + 146P_4 + 77P_5 + 2P_6 + 1P_7. \]

First compute \( P'_4 = P_4 + P_3 \) and then recursively compute
\[ 32P_1 + 16P_2 + 154P_3 + 146P_4' + 77P_5 + 2P_6 + 1P_7. \]

Same scalars show up as before.

Ed25519 batch verification: verify batch of 64 signatures about twice as fast as verifying each separately.
Reduce largest row:

\[
\begin{align*}
000000000 &= 0 \\
000000000 &= 0 \\
000000000 &= 0 \\
000000000 &= 0 \\
000000000 &= 0 \\
000000000 &= 0 \\
000000000 &= 0 \\
000000000 &= 0 \\
\end{align*}
\]

plus 12 additions.

Final addition chain: 1, 2, 3, 5, 8, 16, 32, 37, 69, 77, 146, 154, 300.

Short, no temporary storage, low two-operand complexity.

Revised goal: Compute

\[
32P_1 + 16P_2 + 300P_3 + 146P_4 + 77P_5 + 2P_6 + 1P_7.
\]

First compute \(P'_4 = P_4 + P_3\) and then recursively compute

\[
32P_1 + 16P_2 + 154P_3 + 146P'_4 + 77P_5 + 2P_6 + 1P_7.
\]

Same scalars show up as before.

Ed25519 batch verification:
verify batch of 64 signatures about twice as fast as verifying each separately.