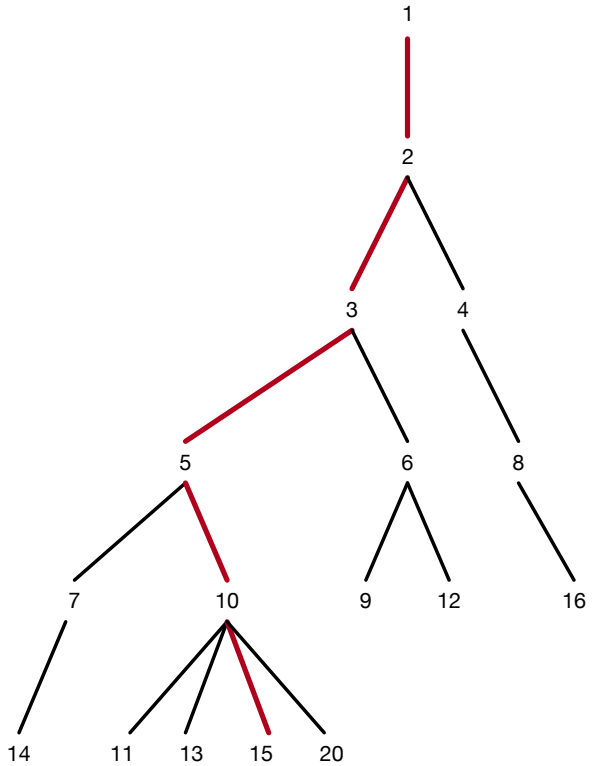
Exponentiation and Point Multiplication



Contents

Exponentiation and Point Multiplication

- Given the integer d, the computation of $S = M^d \pmod{n}$ is the exponentiation operation
- If the modulus n is a prime p, the exponentiation is performed over GF(p) arithmetic, otherwise, n is a composite integer and the exponentiation is performed in the ring $\mathcal{Z}_n$ arithmetic

- On the other hand, the computation of $Q = [d]P$ using a point P in an elliptic curve group $\mathcal{E}$ is a point multiplication operation

- These two operations form the basis of almost all cryptographic algorithms and protocols, such RSA, DH, ElGamal, DSA, ECDH, ECDH, ECIES, EdDSA, and more

Exponentiation and Point Multiplication

- Interestingly and surprisingly, both computations $S = M^d \pmod{n}$ and $Q = [d]P$ are accomplished using similar exponentiation heuristics (algorithms)
- The objective of these hueristics is to use as few modular multiplications or as few elliptic curve additions as possible to compute $M^d \pmod{n}$ or $[d]P$, for a given integer d
- These exponentiation algorithms are also called *addition chains* or *addition-subtraction chains*

Exponentiation Heuristics

- An **addition chain** is a sequence of integers

$$a_0 \quad a_1 \quad a_2 \quad \cdots \quad a_r$$

starting from $a_0 = 1$ and ending with $a_r = d$ such that any a_k is the sum of two earlier integers a_i and a_j in the chain:

$$a_k = a_i + a_j \quad \text{for } 0 < i, j < k$$

- Example: $d = 55$

1	2	3	6	12	13	26	27	54	55
1	2	3	6	12	13	26	52	55	
1	2	4	5	10	20	40	50	55	
1	2	3	5	10	11	22	44	55	

Addition Chains

- We should prefer shorter chains
- Consider the addition chain for $d = 55$

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 10 \rightarrow 11 \rightarrow 22 \rightarrow 44 \rightarrow 55$$

- It yields an algorithm for computing $S = M^d \pmod{n}$

$$M^1 \rightarrow M^2 \rightarrow M^3 \rightarrow M^5 \rightarrow M^{10} \rightarrow M^{11} \rightarrow M^{22} \rightarrow M^{44} \rightarrow M^{55}$$

- Similarly, for computing $Q = [d]P$

$$P \rightarrow [2]P \rightarrow [3]P \rightarrow [5]P \rightarrow [10]P \rightarrow [11]P \rightarrow [22]P \rightarrow [44]P \rightarrow [55]P$$

- The length of the chain:
  The number of multiplications+squarings to compute $M^d \pmod{n}$
  The number of point additions+doublings to compute $[d]P$

Addition Chains

- Finding the shortest addition chain for a given (unbounded) d is an NP-complete problem
- Upper bound: $\lfloor \log_2 d \rfloor + H(d) - 1$, where $H(d)$ is the Hamming weight of d (the binary method)
- Lower bound: $\log_2 d + \log_2 H(d) - 2.13$ (proven by Schönhage)
- Addition chain algorithms: Power tree, factor, binary, m-ary, adaptive m-ary, and sliding windows methods
- Optimization techniques, such as simulated annealing, were used to produce short addition chains for certain exponents

Power Tree Method

- The power tree method creates a tree of all powers M up to d
- It recursively builds new chains from the existing ones
- Therefore, it always finds the shortest chains
- Finding the shortest chain is an NP-complete problem
- The power tree method requires exponential time and space in k
- The power tree method may be useful for small exponents

Power Tree Construction

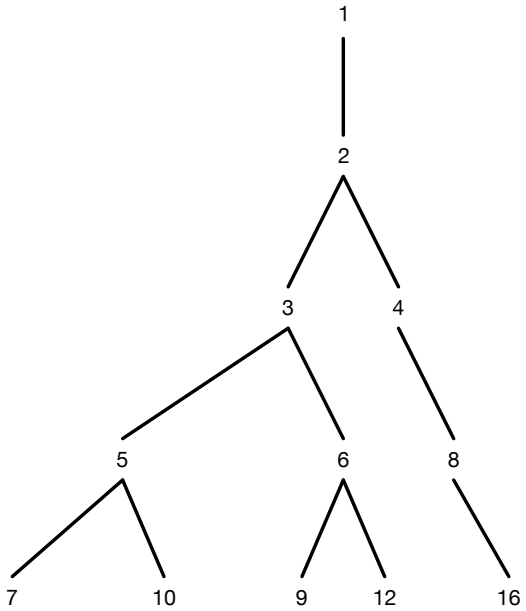
- The root of the tree has 1
- Scan the nodes of the tree at the mth level, from left to right
- Consider the node e at the the mth level
- Construct the $(m+1)$st level of the tree by attaching below the node e the nodes with values

$$e + a_1 \quad e + a_2 \quad e + a_3 \quad \cdots \quad e + a_r$$

  where $a_1, a_2, \ldots, a_r$ is the path from the root of the tree to the node e, therefore, $a_1 = 1$ and $a_r = e$
- Discard duplicate notes that have already appeared in the tree

Power Tree Construction

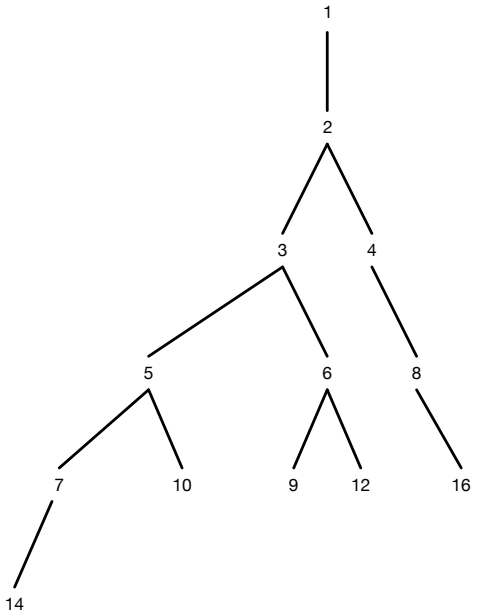
- Consider the power tree whose 4 levels are already completed



- We start with the leftmost node 7
- Path from root to 7 visits 1, 2, 3, 5, 7
- Possible new nodes:
  $7 + 1 = 8$
  $7 + 2 = 9$
  $7 + 3 = 10$
  $7 + 5 = 12$
  $7 + 7 = 14$
- Nodes 8, 9, 10, and 12 are in tree
- We add only 14 below 7

Power Tree Construction

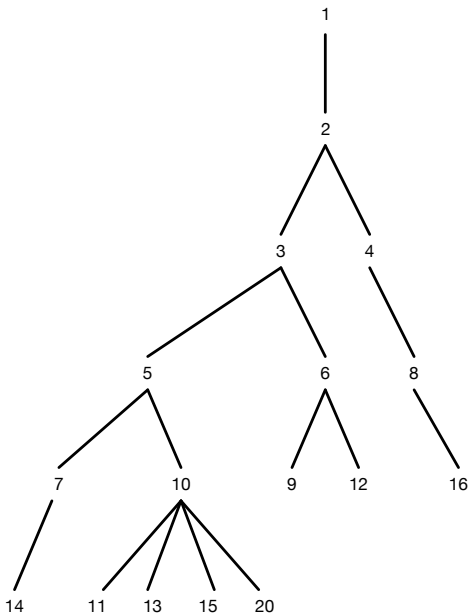
- Therefore, the new power tree would be



- Next node to consider node is 10
- Path from root to 10 visits 1, 2, 3, 5, 10
- Possible new nodes:
  $10 + 1 = 11$
  $10 + 2 = 12$
  $10 + 3 = 13$
  $10 + 5 = 15$
  $10 + 10 = 20$
- Node 12 is in tree
- We add 11, 13, 15, 20 below 10

Power Tree Construction

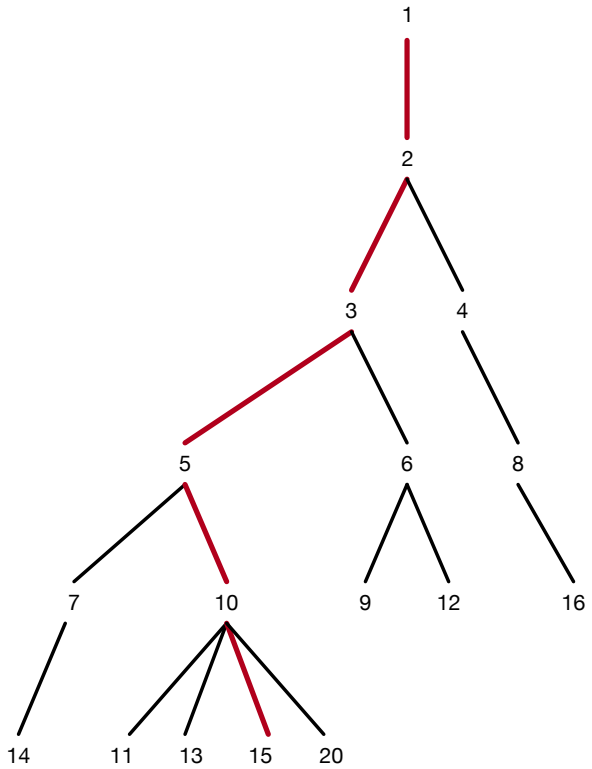
- Therefore, the new power tree would be



- Next node to consider node is 9
- Path from root to 10 visits 1, 2, 3, 6, 9
- Possible new nodes:
  $9 + 1 = 10$
  $9 + 2 = 11$
  $9 + 3 = 12$
  $9 + 6 = 15$
  $9 + 9 = 18$
- Nodes 10, 11, 12, 14 are in tree
- We add only 18 below 9

Computation using the Power Tree

- Once the power tree is constructed, we need to find exponent d in it
- The power tree requires exponential time and space in terms of k
- If d is in tree, the path from the root to d is the **shortest chain**



- The chain for $d = 15$
- $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 10 \rightarrow 15$
- The length of the chain is 5
- 5 is the shortest chain length for $d = 15$

Computation using the Power Tree

- For $d = 15$, the power tree produces the length 5 chain

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 10 \rightarrow 15$$

- This is the shortest chain length for $d = 15$
- The binary method produces a chain of length 6:

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 14 \rightarrow 15$$

- There may be other addition chains of length 5 for $d = 15$
- However, no chain exists with length less than 5

Various Shortest Chains by Power Tree

d	Length	Shortest Chain
15	5	1, 2, 3, 5, 10, 15
55	8	1, 2, 3, 5, 10, 11, 22, 44, 55
119	9	1, 2, 3, 5, 7, 14, 28, 56, 112, 119
250	10	1, 2, 3, 5, 10, 15, 25, 50, 75, 125, 250
249	10	1, 2, 3, 5, 10, 20, 40, 43, 83, 166, 249
251	11	1, 2, 3, 5, 7, 14, 28, 31, 62, 124, 248, 251
365	11	1, 2, 3, 5, 10, 15, 30, 45, 90, 180, 360, 365
2730	14	1, 2, 3, 5, 7, 14, 21, 42, 84, 168, 336, 672, 1344, 1365, 2730
3038	10	1, 2, 3, 5, 7, 14, 21, 42, 63, 126, 189, 378, 756, 1512, 1519, 3038
3665	15	1, 2, 3, 5, 7, 14, 28, 56, 112, 224, 229, 458, 916, 1832, 3664, 3665

Factor Method

- The factor method is based on factorization of the exponent $d = a \cdot b$, where a is the smallest prime factor of d and $b > 1$
- We then compute M^d by first computing M^a and then raising this value to the bth power

$$M^d = M^{ab} = (M^a)^b$$

- If d is prime, we first compute M^{d-1} using the factor method, and the multiply by M
- The algorithm proceeds recursively until small exponents are obtained, at which time the power tree method can be utilized

Factor Method Example

- Given $d = 55 = 5 \cdot 11$
- First compute M^5 using $M \rightarrow M^2 \rightarrow M^4 \rightarrow M^5$
- Now assign $x \leftarrow M^5$ and compute x^{11} which would give M^{55}
- The computation of x^{11} is accomplished by first computing x^{10} and then multiplying by x since 11 is prime
- $10 = 2 \cdot 5$ implies we first compute x^2 using $x \rightarrow x^2$
- Now assign $y = x^2$ and compute y^5 using $y \rightarrow y^2 \rightarrow y^4 \rightarrow y^5$
- Finally, to compute x^{11} we use $x^{10} \rightarrow x^{11}$
- Therefore, the factor method computes M^{55} using

$$M \rightarrow M^2 \rightarrow M^4 \rightarrow M^5 \Rightarrow x \rightarrow x^2 \Rightarrow y \rightarrow y^2 \rightarrow y^4 \rightarrow y^5 \Rightarrow x^{10} \rightarrow x^{11} \Rightarrow M^{55}$$

Factor Method Comparison

- The **factor method** produces a **length 8** chain for $d = 55$

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 10 \rightarrow 20 \rightarrow 40 \rightarrow 50 \rightarrow 55$$

- Since $55 = (110111)$, the **binary method** produces a **length 9** chain

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 13 \rightarrow 26 \rightarrow 27 \rightarrow 54 \rightarrow 55$$

- The quaternary method partitions d as $\underline{11}\ \underline{01}\ \underline{11}$
- Preprocessing: $M \rightarrow M^2 \rightarrow M^3$
- Exponentiation: $M^3 \rightarrow M^6 \rightarrow M^{12} \rightarrow M^{13} \rightarrow M^{26} \rightarrow M^{52} \rightarrow M^{55}$
- Thus, the **quaternary method** produces a **length 8** chain

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 13 \rightarrow 26 \rightarrow 52 \rightarrow 55$$

Binary Method

- Known since antiquity: Scan the bits of d and perform squaring and multiplication operation to compute $S = M^d \pmod{n}$
- Similarly: Scan the bits of d and perform point doubling and point addition operation to compute $Q = [d]P$
- The exponent d is assumed to have $k = \lceil \log_2(d) \rceil$ bits
- Generally $d_{k-1} = 1$ but this is not necessary
- There are 2 versions of the binary method: **Left-to-Right** (LR) and **Right-to-Left** (RL) depending on how exponent bits are scanned
- The binary method can be generalized to the m-ary method by scanning several bits at a time

LR Binary Method of Exponentiation

Input: M, d, n
Output: $S = M^d \bmod n$
1: **if** $d_{k-1} = 1$ **then** $S \leftarrow M$ **else** $S \leftarrow 1$
2: **for** $i = k - 2$ **downto** 0
2a: $S \leftarrow S \cdot S \pmod{n}$
2b: **if** $d_i = 1$ **then** $S \leftarrow S \cdot M \pmod{n}$
3: **return** S

LR Binary Method of Exponentiation Example

$d = 55 = (110111)$ and $k = 6$

Step 1: $d_5 = 1 \Longrightarrow S \leftarrow M$

Steps 2a and 2b:

i	d_i	Step 2a (S)	Step 2b (S)
4	1	$(M)^2 = M^2$	$M^2 \cdot M = M^3$
3	0	$(M^3)^2 = M^6$	M^6
2	1	$(M^6)^2 = M^{12}$	$M^{12} \cdot M = M^{13}$
1	1	$(M^{13})^2 = M^{26}$	$M^{26} \cdot M = M^{27}$
0	1	$(M^{27})^2 = M^{54}$	$M^{54} \cdot M = M^{55}$

Step 3: Return $S = M^{55}$

LR Binary Method of Point Multiplication

Input: P, d
Output: $Q = [d]P$
1: **if** $d_{k-1} = 1$ **then** $Q \leftarrow P$ **else** $Q \leftarrow \mathcal{O}$
2: **for** $i = k - 2$ **downto** 0
2a: $Q \leftarrow Q \oplus Q$
2b: **if** $d_i = 1$ **then** $Q \leftarrow Q \oplus P$
3: **return** Q

LR Binary Method of Point Multiplication Example

$d = 55 = (110111)$ and $k = 6$

Step 1: $d_5 = 1 \Longrightarrow Q \leftarrow P$

Steps 2a and 2b:

i	d_i	Step 2a (Q)	Step 2b (Q)
4	1	$P \oplus P = [2]P$	$[2]P \oplus P = [3]P$
3	0	$[3]P \oplus [3]P = [6]P$	$[6]P$
2	1	$[6]P \oplus [6]P = [12]P$	$[12]P \oplus P = [13]P$
1	1	$[13]P \oplus [13]P = [26]P$	$[26]P \oplus P = [27]P$
0	1	$[27]P \oplus [27]P = [54]P$	$[54]P \oplus P = [55]P$

Step 3: Return $Q = [55]P$

RL Binary Method of Exponentiation

Input: M, d, n
Output: $S = M^d \bmod n$
1: $S \leftarrow 1$ and $T \leftarrow M$
2: **for** $i = 0$ **to** $k - 2$
2a: **if** $d_i = 1$ **then** $S \leftarrow S \cdot T \pmod{n}$
2b: $T \leftarrow T \cdot T \pmod{n}$
3: **if** $d_i = 1$ **then** $S \leftarrow S \cdot T \pmod{n}$
4: **return** S

RL Binary Method of Exponentiation Example

$d = 55 = (110111)$ and $k = 6$

Step 1: $S \leftarrow 1$ and $T \leftarrow M$

Steps 2a and 2b:

i	d_i	Step 2a (S)	Step 2b (T)
0	1	$1 \cdot M = M$	$(M)^2 = M^2$
1	1	$M \cdot M^2 = M^3$	$(M^2)^2 = M^4$
2	1	$M^3 \cdot M^4 = M^7$	$(M^4)^2 = M^8$
3	0	M^7	$(M^8)^2 = M^{16}$
4	1	$M^7 \cdot M^{16} = M^{23}$	$(M^{16})^2 = M^{32}$

Step 3: $d_5 = 1 \implies S \leftarrow S \cdot T = M^{23} \cdot M^{32} = M^{55}$

Step 4: Return $S = M^{55}$

RL Binary Method of Point Multiplication

Input: P, d
Output: $Q = [d]P$
1: $Q \leftarrow \mathcal{O}$ and $R \leftarrow P$
2: **for** $i = 0$ **to** $k - 2$
2a: **if** $d_i = 1$ **then** $Q \leftarrow Q \oplus R$
2b: $R \leftarrow R \oplus R$
3: **if** $d_i = 1$ **then** $Q \leftarrow Q \oplus R$
4: **return** Q

RL Binary Method of Point Multiplication

$d = 55 = (110111)$ and $k = 6$

Step 1: $Q \leftarrow \mathcal{O}$ and $R \leftarrow P$

Steps 2a and 2b:

i	d_i	Step 2a (Q)	Step 2b (R)
0	1	$\mathcal{O} \oplus P = P$	$P \oplus P = [2]P$
1	1	$P \oplus [2]P = [3]P$	$[2]P \oplus [2]P = [4]P$
2	1	$[3]P \oplus [4]P = [7]P$	$[4]P \oplus [4]P = [8]P$
3	0	$[7]P$	$[8]P \oplus [8]P = [16]P$
4	1	$[7]P \oplus [16]P = [23]P$	$[16]P \oplus [16]P = [32]P$

Step 3: $d_5 = 1 \implies Q \leftarrow Q \cdot R = [23]P \oplus [32]P = [55]P$

Step 4: Return $Q = [55]P$

The m-ary Method

- By scanning the bits of d
  - 2 at a time: The quaternary method
  - 3 at a time: The octal method
  - ...
  - w at a time: The m-ary method ($m = 2^w$)
- We may need all powers M^v or $[v]P$ for $v \in [0, 2^w - 1]$
- At each step 2 squaring operations performed
- It is also called the "window" method
- The width of the window is w bits
- Consider the **quaternary** method: $d = 250 = \underline{11}\ \underline{11}\ \underline{10}\ \underline{10}$

Quaternary Exponentiation Example

- $d = 250 = \underline{11}\ \underline{11}\ \underline{10}\ \underline{10}$
- Preprocessing

bits	v	M^v
00	0	1
01	1	M
10	2	$M \cdot M = M^2$
11	3	$M^2 \cdot M = M^3$

- Starting value $S = M^3$ since the leftmost window is $(11)_2$

bits	Step 2a (S)	Step 2b (S)
11	$(M^3)^4 = M^{12}$	$M^{12} \cdot M^3 = M^{15}$
10	$(M^{15})^4 = M^{60}$	$M^{60} \cdot M^2 = M^{62}$
10	$(M^{62})^4 = M^{248}$	$M^{248} \cdot M^2 = M^{250}$

- The quaternary method requires $2 + 6 + 3 = 11$ multiplications

Quaternary Point Multiplication Example

- $d = 250 = \underline{11}\ \underline{11}\ \underline{10}\ \underline{10}$
- Preprocessing

bits	v	$[v]P$
00	0	$\mathcal{O}$
01	1	P
10	2	$P \oplus P = [2]P$
11	3	$P \oplus [2]P = [3]P$

- Starting value $Q = [3]P$ since the leftmost window is $(11)_2$

bits	Step 2a (Q)	Step 2b (Q)
11	$[4]([3]P) = [12]P$	$[12]P \oplus [3]P = [15]P$
10	$[4]([15]P) = [60]P$	$[60]P \oplus [2]P = [62]P$
10	$[4]([62]P) = [248]P$	$[248]P \oplus [2]P = [250]P$

- The quaternary method requires $2 + 6 + 3 = 11$ point additions

The m-ary Method

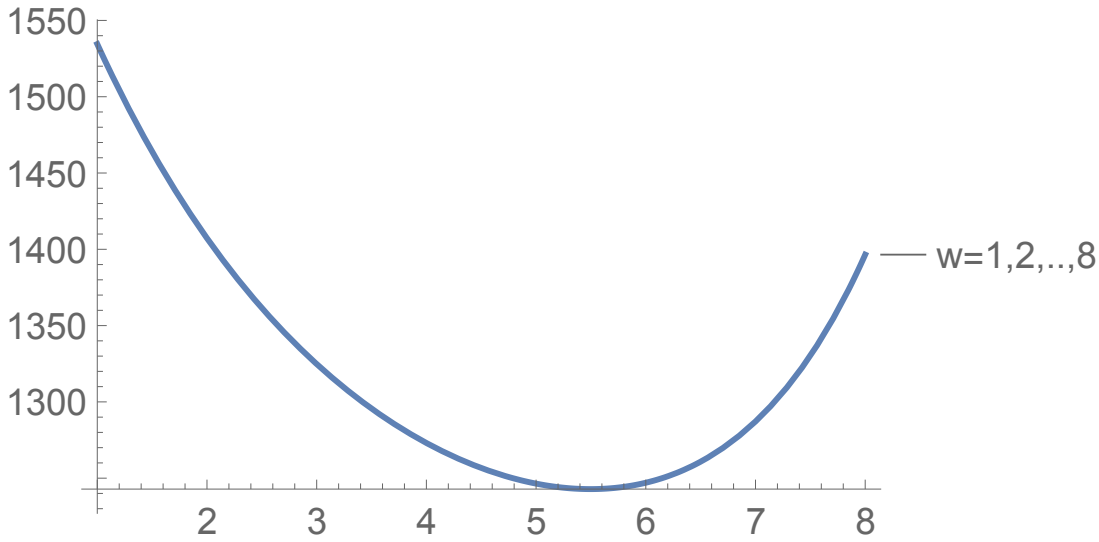
- The average number of multiplications for the m-ary method:
  - Preprocessing multiplications: $2^w - 2$
    This is for computing $M^2, M^3, M^4, \ldots, M^{2^w-1}$
  - Squarings: $(\frac{k}{w} - 1) \cdot w = k - w$
    There are $(k/w - 1)$ digits
    For each digit we perform w squaring operations
  - Multiplications: $\frac{m-1}{m} \cdot (\frac{k}{w} - 1) = (1 - 2^{-w})(k - w)/w$
    There are $(k/w - 1)$ digits
    If the digit is zero (probability $1/m$), we do not perform a multiplication
    If the digit is not zero (probability $(m-1)/m$), we perform a multiplication
- Total number of multiplications and squarings

$$T(k, w) = 2^w - 2 + k - w + (1 - 2^{-w})(k - w)/w$$

where w is the window length, $w = 1, 2, 3, \ldots$

The m-ary Method

- The average number of multiplications for a given k is a function of the window length w
- For example, for $k = 1024$ and the window length as $w = 1, 2, \ldots, 8$, we find the average number of multiplications as



- There is an optimal w for every k, which is not necessarily an integer

The m-ary Method

- The average number of multiplications and the optimal value of (integer) w for $k = 256, 512, 1024, 2048$

k	BM	MM	w^*	Savings %
256	383	325	4	15.1
512	767	635	5	17.2
1024	1535	1246	5	18.8
2048	3071	2439	6	20.6

k	w^*	MM/BM
10^4	8	0.765882
10^8	19	0.705155
10^{16}	42	0.682753
10^{32}	93	0.673889

- Asymptotic value of MM/BM is $\frac{2}{3}$ as $k \to \infty$

Reducing the Preprocessing Cost

- The m-ary method with w-bit window computes all powers of $M^i \pmod{n}$ for $i = 2, 3, \ldots, 2^w - 1$
- This requires $2^w - 2$ multiplications
- However, not all 2^w window bit configurations may appear
- For example, consider the following exponent with $k = 16$ and $w = 4$

$$\underline{1011}\ \underline{0011}\ \underline{0111}\ \underline{1000}$$

- These windows impliy that we need to compute $M^v \pmod{n}$ for only $v = 3, 7, 8, 11$ during preprocessing

Reducing the Preprocessing Cost

- We can judiciously compute the powers $M^v \pmod{n}$ for only $v = 3, 7, 8, 11$ using

$$\begin{aligned} M^2 &\leftarrow M \cdot M \\ M^3 &\leftarrow M^2 \cdot M \\ M^4 &\leftarrow M^2 \cdot M^2 \\ M^7 &\leftarrow M^3 \cdot M^4 \\ M^8 &\leftarrow M^4 \cdot M^4 \\ M^{11} &\leftarrow M^3 \cdot M^8 \end{aligned}$$

- This requires 6 multiplications
- Computing all possible exponents would require $2^4 - 2 = 14$ multiplications during preprocessing

Sliding Window Algorithms

- Based on data-dependent m-ary partitioning of the exponent
- Constant-Length Nonzero Windows (CLNW):
  Rule: Partition the exponent into zero words of any length and nonzero words of constant length w
- Variable-Length Nonzero Windows (VLNW):
  Rule: Partition the exponent into zero words of length at least q and nonzero words of length at most w

Constant-Length Sliding Window Algorithm

- The partitioning starts from the LSB and moves to the MSB
- If the first bit of a window is 1, it is considered as a nonzero window and w bits taken into a constant-length window
- If the first bit of a window is 0, this bit an all adjacent zeros are taken into a variable-length zero window
- Example: $d = (111001010001)$ for $w = 3$ as
- Partitioning: $\underline{111}\ \underline{00}\ \underline{101}\ \underline{0}\ \underline{001}$
- Example: $d = (101000000100001111)$ for $w = 3$
- Partitioning: $\underline{101}\ \underline{00000}\ \underline{001}\ \underline{00}\ \underline{001}\ \underline{111}$

Constant-Length Sliding Window Algorithm

- Consider $d = 3665 = \underline{111}\ \underline{00}\ \underline{101}\ \underline{0}\ \underline{001}$
- First we compute $M^v \pmod n$ for the given window values using as few multiplications as possible
- Note that since the first bit every nonzero window is always 1, we need to compute only for odd v values
- In this case, we have $v = 1, 5, 7$, which can be computed as

$$\begin{aligned} M^2 &\leftarrow M \cdot M \\ M^3 &\leftarrow M^2 \cdot M \\ M^5 &\leftarrow M^3 \cdot M^2 \\ M^7 &\leftarrow M^5 \cdot M^2 \end{aligned}$$

- This requires 4 multiplications
- In the worst case, we need to perform $1 + \frac{2^w - 2}{2}$ multiplications

Constant-Length Sliding Window Algorithm

- The algorithm uses the precomputed window values, and computes $M^d \pmod n$ by performing w consecutive squaring operations when a nonzero window is scanned, skipping over zeros and performing a squaring for every zero when a zero window is scanned
- $d = 3665 = \underline{111}\ \underline{00}\ \underline{101}\ \underline{0}\ \underline{001}$

bits	Step 2a	Step 2b
111	M^7	M^7
00	$(M^7)^4 = M^{28}$	M^{28}
101	$(M^{28})^8 = M^{224}$	$M^{224} \cdot M^5 = M^{229}$
0	$(M^{229})^2 = M^{458}$	M^{458}
001	$(M^{458})^8 = M^{3664}$	$M^{3664} \cdot M^1 = M^{3665}$

Constant-Length Sliding Window Algorithm

- The CLNW method reduces the average cost of the exponentiation another 3-6% compared to the m-ary method

k	m-ary	w^*	CLNW	w^*	%
128	167	4	156	4	6.6
256	325	4	308	5	5.2
512	635	5	607	5	4.4
1024	1246	5	1195	6	4.1
2048	2439	6	2360	7	3.2

Variable-Length Sliding Window Algorithm

- The partitioning starts from the LSB and moves to the MSB
- If the first bit of a window is 1, it is considered as a nonzero window and **up to** w bits taken into a constant-length window, until a zero is observed
- If the first bit of a window is 0, this bit an all adjacent **at least** q zeros are taken into a variable-length zero window
- Examples for $w = 5$ and $q = 2$

$$\begin{array}{r} \underline{101}\ \underline{0}\ \underline{11101}\ \underline{00}\ \underline{101} \\ \underline{10111}\ \underline{000000}\ \underline{1}\ \underline{00}\ \underline{111}\ \underline{000}\ \underline{1011} \end{array}$$

- Examples for $w = 10$ and $q = 4$

$$\begin{array}{r} \underline{1011011}\ \underline{0000}\ \underline{11}\ \underline{0000} \\ \underline{11110111}\ \underline{00}\ \underline{1111110101}\ \underline{0000}\ \underline{11011} \end{array}$$

Variable-Length Sliding Window Algorithm

- The VLNW method reduces the average cost of the exponentiation another 5-8% compared to the m-ary methid

	m-ary		VLNW		for q^*
k	T/k	w^*	T/k	w^*	%
128	1.31	4	1.20	4	7.8
256	1.27	4	1.18	4	6.8
512	1.24	5	1.16	5	6.4
1024	1.22	5	1.15	6	5.8
2048	1.19	6	1.13	6	5.0

Addition-Subtraction Chains

- An addition-subtraction chain is a sequence of integers

$$a_0 \quad a_1 \quad a_2 \quad \cdots \quad a_r$$

starting from $a_0 = \pm 1$ and ending with $a_r = d$ such that any a_k is the sum or the difference of two earlier integers a_i and a_j in the chain:

$$a_k = a_i \pm a_j \quad \text{for } 0 < i, j < k$$

- Example: $d = 55$

$$\pm 1 \quad 2 \quad 4 \quad 8 \quad 7 \quad 14 \quad 28 \quad 56 \quad 55$$

- An addition-subtraction chain yields an algorithm for computing $M^d \pmod n$ or $[d]P$ given the integer d

Canonical Encoding algorithm

- The canonical encoding algorithm is an efficient way to generate addition-subtraction chains
- The canonical encoding algorithm computes the sparse signed-digit representation of the exponent with digits $\{0, 1, -1\} = \{0, 1, \bar{1}\}$
- It uses the property that a trail of adjacent 1s can be replaced with a single 1 followed by 0s and then a single $-1 = \bar{1}$ digit
- The property is based on the identity that for $i > j$

$$\begin{aligned} 2^i + 2^{i-1} + \cdots + 2^{j+1} + 2^j &= 2^{i+1} - 2^j \\ (111 \cdots 111) &= (1000 \cdots 00\bar{1}) \end{aligned}$$

- The resulting encoded number may be 1 bit longer

Canonical Encoding Algorithm

- An example of the signed-digit representation

$$\begin{aligned} 30 = (011110) &= 2^4 + 2^3 + 2^2 + 2^1 \\ 30 = (100\bar{1}0) &= 2^5 - 2^1 \end{aligned}$$

- Thus, we represent the integer using fewer nonzero bits
- Arithmetic with signed-digit encoded integers have some advantages
- However, we must deal with the negative digit $\bar{1}$ within the exponentiation algorithm

Canonical Encoding Algorithm

- The variable c_i is temporary, with starting value $c_0 = 0$
- The encoding proceeds from the LSB d to the MSB

d_{i+1}	d_i	c_i	f_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	**0**	**0**
1	0	1	$\bar{1}$	1
1	1	0	$\bar{1}$	1
1	1	1	**0**	**1**

Encoding of $d = 3038$

d	0	1	0	1	1	1	1	0	1	1	1	1	0	
f	1	0	$\bar{1}$	0	0	0	0	$\bar{1}$	0	**0**	0	$\bar{1}$	**0**	
c	0	1	1	1	1	1	1	1	1	**1**	1	1	**0**	0

Canonical Encoding Algorithm

- The canonical encoding algorithm **optimally** encodes the integer exponent using the digit set $\{0, 1, \bar{1}\}$
- It produces an encoding of d with minimal number of nonzero digits

Theorem

Given a k-bit integer d, the Canonical Encoding Algorithm generates its encoding using the digit set $\{0, 1, \bar{1}\}$ in such a way that its average number of nonzero digits is minimal and it is equal to $k/3$.

- A canonically encoded exponent never has two adjacent nonzero bits (such as 11 or $\bar{1}1$ or $1\bar{1}$)
- Such representations are also called Non-Adjacent Forms (NAFs)

The NAF (Non-Adjacent Form) Algorithm

- There is another and **equivalent** algorithm for computing the canonical encoding of d, referred as the NAF algorithm

NAF Algorithm
Input: d
Output: $f = \text{NAF}(d)$
1: $i \leftarrow 0$
2: **while** $d \geq 1$
2a: **if** d is odd
2a1: $f_i \leftarrow 2 - (d \bmod 4)$
2a2: $d \leftarrow d - f_i$
2b: **else**
2b1: $f_i \leftarrow 0$
2c: $i \leftarrow i + 1$
2d: $d \leftarrow d/2$
3: **return** f

NAF Algorithm Example

- Given $d = 3038$, the computation of $f = \text{NAF}(d)$

i	d	d is odd		d is even	new
		f_i	d	f_i	d
0	3038			0	1519
1	1519	-1	1520		1520
2	1520			0	760
3	760			0	380
4	380			0	190
5	190			0	95
6	95	-1	96		96
7	48			0	24
8	24			0	12
9	12			0	6
10	6			0	3
11	3	-1	4		2
12	2			0	1
13	1	1	0		0

- The result $f = (10\bar{1}000\bar{1}000\bar{1}0)$

Canonical Encoding Binary Exponentiation

- The number of squaring operations in the standard binary method of exponentiation is equal to the number of bits in d
- On the other hand, the number of multiplication operations in the standard binary method of exponentiation is equal to the number of 1s in d, that is, its Hamming weight
- Therefore, the standard binary method requires an average of k squaring and $\frac{k}{2}$ multiplication operations
- The canonical encoding binary method however uses the encoded exponent f which has $2k/3$ zero bits and $k/3$ nonzero bits
- Therefore, the canonical encoding binary method k squaring and $\frac{k}{3}$ multiplication operations
- However, it also requires $M^{-1} \pmod{n}$ as input

Canonical Encoding Binary Exponentiation

Canonical Encoding Binary Method

Input: M, M^{-1}, d, n

Output: $S = M^d \bmod n$

0: Obtain signed-digit encoding f of d

1: **if** $f_{k-1} = 1$ **then** $S \leftarrow M$ **else** $S \leftarrow 1$

2: **for** $i = k-2$ **downto** 0

2a: $S \leftarrow S \cdot S \pmod{n}$

2b: **if** $f_i = 1$ **then** $S \leftarrow S \cdot M \pmod{n}$

if $f_i = \bar{1}$ **then** $S \leftarrow S \cdot M^{-1} \pmod{n}$

3: **return** S

- This method for computing $M^d \pmod{n}$ requires $M^{-1} \pmod{n}$
- This is as costly as computing the exponentiation $M^d \pmod{n}$

Addition-Subtraction Chains for Point Multiplication

- The application of any addition-subtraction chain requires the computation of the inverse (negative) of a point
- For example, this addition-subtraction chain

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 15$$

computes $[15]P$ by first computing

$$P \rightarrow [2]P \rightarrow [4]P \rightarrow [8]P \rightarrow [16]P \rightarrow [15]P$$

- However, at the last step we need to compute $[16]P \oplus (-P)$, which requires the availability of $-P$
- In elliptic curves, the inverse of a point is easily computed

Canonical Encoding Binary Point Multiplication

- For elliptic curves over GF(p): if $P = (x, y)$, then $-P = (x, -y)$
- For elliptic curves over GF(2^k): if $P = (x, y)$, then $-P = (x, x + y)$

Input: $P, -P, d$
Output: $Q = [d]P$
0: Obtain signed-digit encoding f of d
1: **if** $f_{k-1} = 1$ **then** $Q \leftarrow P$ **else** $Q \leftarrow \mathcal{O}$
2: **for** $i = k - 2$ **downto** 0
2a: $Q \leftarrow Q \oplus Q$
2b: **if** $f_i = 1$ **then** $Q \leftarrow Q \oplus P$
2c: **if** $f_i = \bar{1}$ **then** $Q \leftarrow Q \oplus (-P)$
3: **return** Q

Canonical Encoding Binary Point Multiplication

- Exponent: $d = 119 = (1110111)$
- Canonically encoded exponent: $f = 1000\bar{1}00\bar{1}$
- We start with $Q = P$ since $f_7 = 1$

i	f_i	Step 2a (Q)	Step 2b or 2c (Q)
6	0	$[1]P \oplus [1]P = [2]P$	$[2]P$
5	0	$[2]P \oplus [2]P = [4]P$	$[4]P$
4	0	$[4]P \oplus [4]P = [8]P$	$[8]P$
3	$\bar{1}$	$[8]P \oplus [8]P = [16]P$	$[16]P \oplus (-P) = [15]P$
2	0	$[15]P \oplus [15]P = [30]P$	$[30]P$
1	0	$[30]P \oplus [30]P = [60]P$	$[60]P$
0	$\bar{1}$	$[60]P \oplus [60]P = [120]P$	$[120]P \oplus (-P) = [119]P$

- The binary method requires $6 + 5 = 11$ point additions+doublings
- The canonical method requires $7 + 2 = 9$ point additions+doublings

The m-ary NAF Algorithm

- The Canonical Encoding Algorithm produces the encoding of the exponent with the digit set $\{0, 1, \bar{1}\}$
- However, this algorithm does not extend to higher radix methods
- If we can scan the exponent w bits at a time, similar to the m-ary method for $m = 2^w$, the resulting exponent may no longer be in nan-adjacent form for radix 2^w
- For example, consider the binary exponent $d = 3038$ and its canonical (binary) encoding as $f = (10\bar{1}000\bar{1}000\bar{1}0)$
- If f is scanned 2, 3 or 4 bits at a time, we obtain

$$\begin{aligned} \underline{10}\ \underline{\bar{1}0}\ \underline{00}\ \underline{0\bar{1}}\ \underline{00}\ \underline{00}\ \underline{\bar{1}0} &= 2\bar{2}0\bar{1}00\bar{2} \\ \underline{10}\ \underline{\bar{1}00}\ \underline{00\bar{1}}\ \underline{000}\ \underline{0\bar{1}0} &= 2\bar{4}0\bar{1}0\bar{2} \\ \underline{10}\ \underline{\bar{1}000}\ \underline{0\bar{1}00}\ \underline{00\bar{1}0} &= 2\bar{8}\bar{4}\bar{2} \end{aligned}$$

- Neither one of these encodings are in NAF

The w-width NAF Algorithm

- However, the m-ary version of the NAF algorithm directly and correctly computes the w-width NAF of d for $m = 2^w$

NAF Algorithm
Input: d, w
Output: $f = \text{NAF}_w(d)$
1: $i \leftarrow 0$
2: **while** $d \geq 1$
2a: **if** d is odd
2a1: $f_i \leftarrow d \pmod{2^w}$
2a2: $d \leftarrow d - f_i$
2b: **else**
2b1: $f_i \leftarrow 0$
2c: $i \leftarrow i + 1$
2d: $d \leftarrow d/2$
3: **return** f

The w-width NAF Algorithm Properties

- The m-ary version of the NAF algorithm differs from the binary version only in Step 2a1 where the **smod** (signed mod) operation is used, instead of the usual **mod** (unsigned mod) operation
- The smod operation produces $d \pmod{2^w}$ in the **least magnitude representation**, i.e., in the range $[-2^{w-1}, 2^{w-1})$
- For $w = 3$, the range would be $[-4, 4) = \{-4, -3, -2, -1, 0, 1, 2, 3\}$
- Example: $10 \pmod 8$ is equal to 2, thus, $10 \pmod 8$ gives 2
  Similarly: $13 \pmod 8$ is equal to 5, thus, $13 \pmod 8$ gives -3

The w-width NAF Algorithm Properties

- The length of the w-width NAF encoded exponent f is at most one more than the length of the binary representation d
- All digits f_i of the w-width NAF encoded exponent f is odd, and at most one of w consecutive digits is nonzero
- The w-width NAF Algorithm produces a **binary encoding**, in the sense that while the digits are not binary, but the weights of the digits are in increasing powers of 2
- For $w = 1$, the w-width NAF Algorithm produces the usual binary encoding with digits $\{0, 1\}$
- For $w = 2$, the w-width NAF Algorithm produces the canonical binary encoding with digits $\{-1, 0, 1\}$

The w-width NAF Algorithm Properties

- For all $w \geq 2$ values, the w-width NAF Algorithm produces the w-width NAF binary-weighted encoding of d using **zero** and **odd nonzero** values from the digit set

$$[-2^{w-1}, 2^{w-1}) \;=\; \{-2^{w-1}, -2^{w-1}+1, \ldots, -1, 0, 1, \ldots, 2^{w-1}-1\}$$

- For $w = 2$, the digit set is $[-2, 2)$ and $f_i \in \{-1, 0, 1\}$
- For $w = 3$, the digit set is $[-4, 4)$ and $f_i \in \{-3, -1, 0, 1, 3\}$
- For $w = 4$, the digit set is $[-8, 8)$ and $f_i \in \{-7, -5, -3, -1, 0, 1, 3, 5, 7\}$

The w-width NAF Algorithm Examples

- The w-width NAF encoding of $d = 3038 = (101111011110)$

$$
\begin{array}{lcrcl}
w = 2 & \rightarrow & 10\bar{1}000\bar{1}000\bar{1}0 & = & 2^{12} - 2^{10} - 2^6 - 2^1 \\
w = 3 & \rightarrow & 3000\bar{1}000\bar{1}0 & = & 3 \cdot 2^{10} - 2^5 - 2^1 \\
w = 4 & \rightarrow & 3000\bar{1}000\bar{1}0 & = & 3 \cdot 2^{10} - 2^5 - 2^1 \\
w = 5 & \rightarrow & 10000(15)0000(15)0 & = & 2^{11} + 15 \cdot 2^6 + 15 \cdot 2^1 \\
w = 6 & \rightarrow & 300000000(\overline{17})0 & = & 3 \cdot 2^{10} - 17 \cdot 2^1
\end{array}
$$

- The w-width NAF encoding of $d = 2730 = (101010101010)$

$$
\begin{array}{lcrcl}
w = 2 & \rightarrow & 101010101010 & = & 2^{11} + 2^9 + 2^7 + 2^5 + 2^3 + 2^1 \\
w = 3 & \rightarrow & 300\bar{3}00300\bar{3}0 & = & 3 \cdot 2^{10} - 3 \cdot 2^7 + 3 \cdot 2^4 - 3 \cdot 2^1 \\
w = 4 & \rightarrow & 5000500050 & = & 5 \cdot 2^9 + 5 \cdot 2^5 + 5 \cdot 2^1 \\
w = 5 & \rightarrow & 10000(11)0000(\overline{11})0 & = & 2^{11} + 11 \cdot 2^6 - 11 \cdot 2^1 \\
w = 6 & \rightarrow & (21)00000(21)0 & = & 21 \cdot 2^7 + 21 \cdot 2^1
\end{array}
$$

The w-width NAF Point Multiplication

- First we obtain the w-width NAF encoding f of the exponent d
- Assume that the length of f is k
- The digits of f are either zero or odd numbers in the range $f_i \in [-2^{w-1}, 2^{w-1})$
- During the pre-processing stage we compute $[v]P$ for $v = 1, 3, 5, \ldots, 2^{w-1} - 1$, i.e., only for odd values of v
- We place them in a table T such that the vth row of the table contains the point $T(v) = [v]P$ for $v = 1, 3, 5, \ldots, 2^{w-1} - 1$
- We use these values during the point multiplication algorithm
- Furthermore, we perform a point doubling for every digit of f

The w-width NAF Point Multiplication

- First, compute w-width NAF encoding f of d, with length k
- Then, compute $T(v) = [v]P$ for $v = 1, 3, 5, \ldots, 2^{w-1} - 1$

Input: $P, d, f, T(v)$
Output: $Q = [d]P$
1: $Q \leftarrow \mathcal{O}$
2: **for** $i = k - 1$ **downto** 0
2a: $Q \leftarrow Q \oplus Q$
2b: $v \leftarrow$ abs(f_i)
2c: **if** $f_i > 0$ **then** $Q \leftarrow Q \oplus T(v)$
2d: **if** $f_i < 0$ **then** $Q \leftarrow Q \oplus (-T(v))$
3: **return** Q

The w-width NAF Point Multiplication Example

- Exponent: $d = 2730 = (101010101010)$
- The 3-width NAF encoding $f = (300\bar{3}0030\bar{3}0)$
- The processing requires the computation of $[v]P$ for $v = 1, 3$

i	f_i	Step 2a (Q)	v	Step 2c or 2d (Q)
10	3	$\mathcal{O} \oplus \mathcal{O} = \mathcal{O}$	3	$\mathcal{O} + [3]P = [3]P$
9	0	$[3]P \oplus [3]P = [6]P$	0	
8	0	$[6]P \oplus [6]P = [12]P$	0	
7	$\bar{3}$	$[12]P \oplus [12]P = [24]P$	3	$[24]P \oplus [-3]P = [21]P$
6	0	$[21]P \oplus [21]P = [42]P$	0	
5	0	$[42]P \oplus [42]P = [84]P$	0	
4	3	$[84]P \oplus [84]P = [168]P$	3	$[168]P \oplus [3]P = [171]P$
3	0	$[171]P \oplus [171]P = [342]P$	0	
2	0	$[342]P \oplus [342]P = [684]P$	0	
1	$\bar{3}$	$[684]P \oplus [684]P = [1368]P$	3	$[1368]P \oplus [-3]P = [1365]P$
0	0	$[1365]P \oplus [1365]P = [2730]P$	0	

The w-width NAF Point Multiplication

- The average number of nonzero digits in the w-width NAF encoded k-digit exponent is $\frac{k}{w+1}$
- For example, for $w = 2$, the probability of nonzero digit is equal to $\frac{1}{3}$, which is the canonical binary encoding algorithm
- The pre-processing stage, i.e., the computation of $[v]P$ for $v = 1, 3, 5, \ldots, 2^{w-1} - 1$ requires 1 point doubling and $(2^{w-1} - 2)/2 = 2^{w-2} - 1$ point additions
- Since the length of the NAF exponent is k and the average number of nonzero digits is $\frac{k}{w+1}$, the number of point additions is $\frac{k}{w+1}$
- Therefore, the w-width NAF point multiplication for k-bit NAF exponent f requires $1 + k$ doublings and $2^{w-2} - 1 + \frac{k}{w+1}$ additions

Koblitz Curves and Frobenius Map

- Koblitz curves are of the form

$$y^2 + xy = x^3 + ax^2 + 1 \quad \text{for} \quad \text{for} \quad a \in \{0, 1\}$$

- There is an efficient algorithm for computing point multiplication operation $[d]P$ for a given integer d and point P on a Koblitz curve
- It is based on the Frobenius map $\tau(x, y) = (x^2, y^2)$ which satisfies

$$\begin{aligned} \tau(\tau(x, y)) \oplus [2](x, y) &= [\mu]\tau(x, y) \\ (x^4, y^4) \oplus [2](x, y) &= [\mu](x^2, y^2) \end{aligned}$$

- Here $\mu = (-1)^{1-a} = \pm 1$, in other words:

$$(x^4, y^4) \oplus [2](x, y) = \pm(x^2, y^2)$$

Point Multiplication on Koblitz Curves

- Suppose that we need to compute $[d]P$ for an integer d
- Every integer d has a τ-adic expansion:

$$d = d_0 + d_1\tau + d_2\tau^2 + \cdots + d_r\tau^r \quad \text{for} \quad d_i \in \{0, 1, -1, \}$$

- Here τ is the complex number

$$\frac{\mu + \sqrt{7}j}{2} \quad \text{where} \quad \mu = (-1)^{1-a} = \pm 1$$

- We compute $[d]P$ using the Frobenius map as

$$[d]P = [d_0]P \oplus [d_1]\tau(P) \oplus [d_2]\tau^2(P) \oplus \cdots \oplus [d_r]\tau^r(P)$$

Point Multiplication on Koblitz Curves

- Since τ is a complex number, we are expanding the integer d in terms of a sum the powers of τ, with coefficients $d_i \in \{0, 1, -1, \}$
- The application of τ to a point $P = (x, y)$ is accomplished by squaring the coordinate values $\tau(x, y) = (x^2, y^2)$
- Squaring of a field element is essentially free when GF(2^k) is represented using a normal basis
- More precisely, we replace the (signed) binary expansion of the coefficient d with its (signed) τ-adic expansion
- Given $d_i = \pm 1$, we compute

$$[d_i]\tau^i(P) = [d_i]\tau^i(x, y) = \pm(x^{2^i}, y^{2^i})$$

- Therefore, we compute $[d]P$ by summing the nonzero terms

The τ-adic NAF Expansion

- For example, for $a = 1$ and thus $\mu = 1$, we have

$$\tau^5 - \tau^3 = \left(\frac{1+\sqrt{7}j}{2}\right)^5 - \left(\frac{1+\sqrt{7}j}{2}\right)^3 = 8$$

- Therefore, taking $Q = -P$, we compute $[8]P$ using

$$\begin{aligned} [8]P &= [\tau^5 - \tau^3]P \\ &= [\tau^5]P \oplus [\tau^3]Q \\ &= (x^{2^5}, y^{2^5}) \oplus (x^{2^3}, y^{2^3}) \end{aligned}$$

- The computation of $[8]P$ requires only one point addition
- Terms such as x^{2^i} and y^{2^i} in the normal basis are computed by left-rotating the field element vectors i times

The τ-adic NAF Expansion

- The expansion of the integer d in terms of the complex number τ with coefficients $d_i \in \{0, 1, -1, \}$ is called the τ-adic NAF (non-adjacent form) of the integer d, since no two consecutive terms are nonzero
- A NAF expansion (similar to the canonical encoding) produces a signed-digit expansion with minimal number of nonzero digits
- The use of τ-adic NAF gives a significant reduction in the number of terms in the computation of $[d]P$ for the Koblitz curves

Theorem

Every integer has a unique τ-adic NAF.

- The algorithm is given by J. E. Solinas

The τ-adic NAF Expansion Examples

For $a = 1$, we take $\mu = 1$
For $a = 0$, we take $\mu = -1$

$$d = 8 \quad : \quad [1, 0, -1, 0, 0, 0]$$
$$d = 8 \quad : \quad \tau^5 - \tau^3$$

$$d = 9 \quad : \quad [1, 0, -1, 0, 0, 1]$$
$$d = 9 \quad : \quad \tau^5 - \tau^3 + 1$$

$$d = 31 \quad : \quad [1, 0, 1, 0, -1, 0, -1, 0, 0, 0, 0, -1]$$
$$d = 31 \quad : \quad \tau^{11} + \tau^9 - \tau^7 - \tau^5 - 1$$

```
def taunaf(d,mu):
    c0 = d
    c1 = 0
    s = []
    u = 0
    while(c0 != 0 or c1 != 0):
        if (c0%2==1):
            u = 2-(c0-2*c1)%4
            c0 = c0-u
        else:
            u = 0
        s.append(u)
        nc0 = c1 + mu*c0//2
        nc1 = -c0//2
        c0 = nc0
        c1 = nc1
    s.reverse()
    print(len(s)-1,s)
```