Shitcoin Protocol

Didactic Cryptocoin design.

Import packages

In[1]:=
```
Needs["ProgressMapping`", FileNameJoin[{NotebookDirectory[], "ProgressMapping.wl"}]];
```

Cryptography functions

Description:

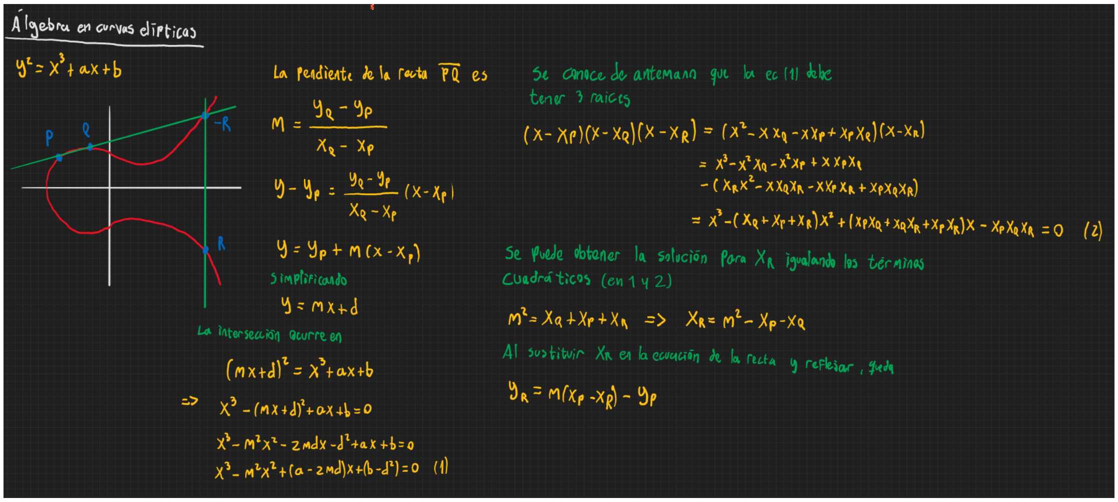
We provide a brief introduction to finite fields. For further information, see Chapter 3 of Koblitz [52], or the books by McEliece [61] and Lidl and Niederreitter [59].
A *finite field* consists of a finite set of elements F together with two binary operations on F, called addition and multiplication, that satisfy certain arithmetic properties. The *order* of a finite field is the number of elements in the field. There exists a finite field of order q if and only if q is a prime power. If q is a prime power, then there is essentially only one finite field of order q; this field is denoted by $\mathbb{F}_q$. There are, however, many ways of representing the elements of $\mathbb{F}_q$. Some representations may lead to more efficient implementations of the field arithmetic in hardware or in software.
If $q = p^m$ where p is a prime and m is a positive integer, then p is called the *characteristic* of $\mathbb{F}_q$ and m is called the *extension degree* of $\mathbb{F}_q$. Most standards which specify the elliptic curve cryptographic techniques restrict the order of the underlying finite field to be an odd prime ($q = p$) or a power of 2 ($q = 2^m$). In §3.1, we describe the elements and the operations of the finite field $\mathbb{F}_p$. In §3.2, elements and the operations of the finite field $\mathbb{F}_{2^m}$ are described, together with two methods for representing the field elements: *polynomial basis representations* and *normal basis representations*.

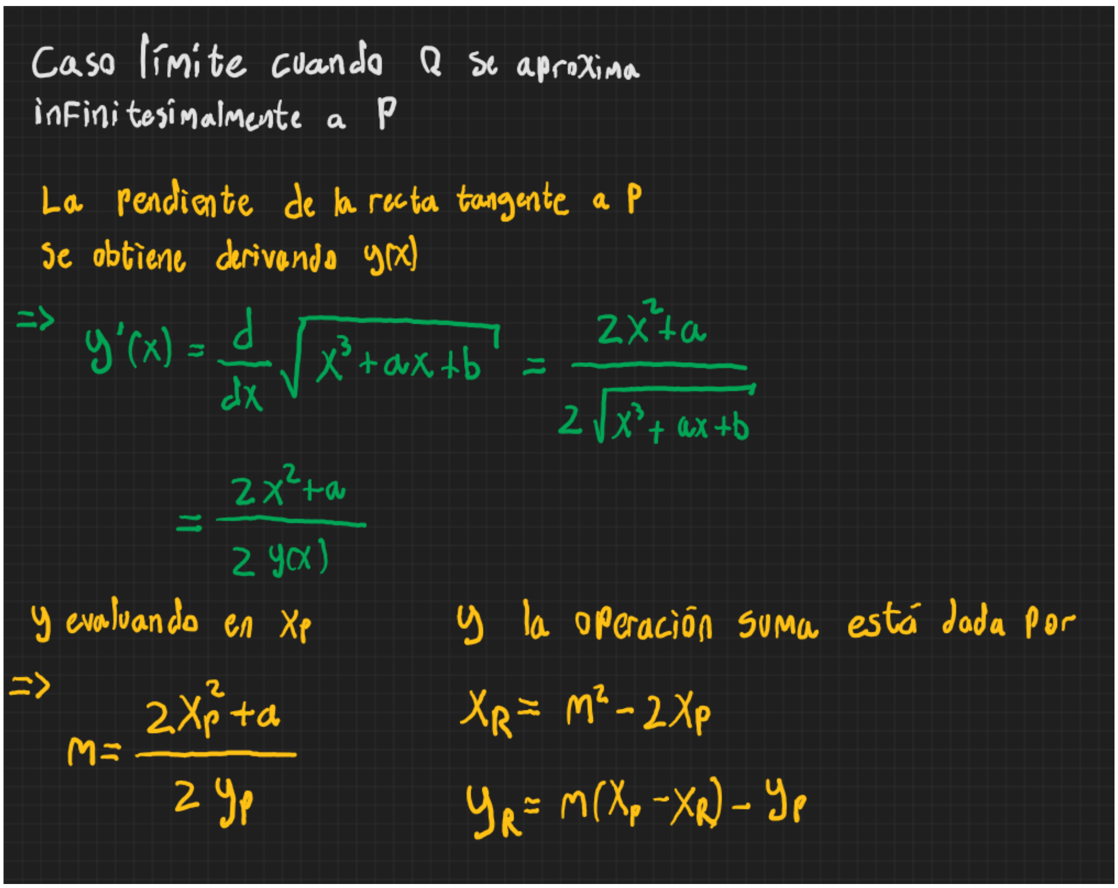
3.1 The Finite Field $\mathbb{F}_p$

Let p be a prime number. The finite field $\mathbb{F}_p$, called a *prime field*, is comprised of the set of integers $\{0, 1, 2, \ldots, p-1\}$ with the following arithmetic operations:

– Addition: If $a, b \in \mathbb{F}_p$, then $a + b = r$, where r is the remainder when $a + b$ is divided by p and $0 \leq r \leq p - 1$. This is known as *addition modulo* p.
– Multiplication: If $a, b \in \mathbb{F}_p$, then $a \cdot b = s$, where s is the remainder when $a \cdot b$ is divided by p and $0 \leq s \leq p - 1$. This is known as *multiplication modulo* p.
– Inversion: If a is a non-zero element in $\mathbb{F}_p$, the *inverse* of a modulo p, denoted a^{-1}, is the unique integer $c \in \mathbb{F}_p$ for which $a \cdot c = 1$.

Example 1. (*The finite field* $\mathbb{F}_{23}$) The elements of $\mathbb{F}_{23}$ are $\{0, 1, 2, \ldots, 22\}$. Examples of the arithmetic operations in $\mathbb{F}_{23}$ are: (i) $12 + 20 = 9$; (ii) $8 \cdot 9 = 3$; and (iii) $8^{-1} = 3$.





Implementation:

In[64]:=

```
EllipticCurveEvaluate[x_, a_, b_] := √(x³ + a x + b);
EllipticCurvePointQ[{x_, y_}, a_, b_, p_] := Block[{lhs, rhs},
   lhs = Mod[y², p];
   rhs = Mod[x³ + a x + b, p];
```

```
    lhs == rhs
   ];
EllipticCurveAdd[{px_, py_}, {qx_, qy_}] := Block[{m, rx, ry},
    If[px == qx, Return[∞]];
    m = (qy - py)/(qx - px);
    rx = m^2 - px - qx;
    ry = m (px - rx) - py;
    {rx, ry}
   ];
EllipticCurveAdd[{px_, py_}, {qx_, qy_}, p_] := Block[{snum, sden, s, rx, ry},
    If[px == qx || ! CoprimeQ[qx - px, p], Return[∞]];

    snum = Mod[qy - py, p];
    sden = ModularInverse[qx - px, p];
    s = Mod[snum * sden, p];
    rx = Mod[s^2 - px - qx, p];
    ry = Mod[s (px - rx) - py, p];
    {rx, ry}
   ];
EllipticCurveDouble[{px_, py_}, a_] := Block[{m, rx, ry},
    m = (3 px^2 + a)/(2 py);
    rx = m^2 - 2 px;
    ry = m (px - rx) - py;
    {rx, ry}
   ];
EllipticCurveDouble[{px_, py_}, a_, p_] := Block[{snum, sden, s, rx, ry},
    snum = Mod[3 px^2 + a, p];
    If[py == 0 || ! CoprimeQ[2 py, p], Return[∞]];
    sden = ModularInverse[2 py, p];
    s = Mod[snum * sden, p];
    rx = Mod[s^2 - 2 px, p];
    ry = Mod[s (px - rx) - py, p];
    {rx, ry}
   ];

ComputeCyclicGroup[G_, a_, p_] := Block[{G2},
    G2 = EllipticCurveDouble[G, a, p];
    Prepend[NestWhileList[EllipticCurveAdd[G, #, p] &, G2, # =!= ∞ &], G]
   ];
MultiplicationPath[G_, a_, p_, k_] := Block[{G2},
    G2 = EllipticCurveDouble[G, a, p];
    Prepend[NestList[EllipticCurveAdd[G, #, p] &, G2, k - 2], G]
   ];
MultiplyBasePoint[G_, a_, p_, k_] := Block[{kBinary, P},
    kBinary = Drop[IntegerDigits[k, 2], 1];
    P = G;
    Do[
     P = EllipticCurveDouble[P, a, p];
```

```
    If[kBinary[[i]] == 1, P = EllipticCurveAdd[P, G, p]];
    ,
    {i, 1, Length[kBinary]}
   ];
   Return[P];
  ];
MultiplyBasePointHex[{hexGx_, hexGy_}, aHex_, pHex_, kHex_] :=
  Block[{Gx, Gy, k, Px, Py, a, p},
   Gx = FromDigits[hexGx, 16];
   Gy = FromDigits[hexGy, 16];
   k = FromDigits[kHex, 16];
   a = FromDigits[aHex, 16];
   p = FromDigits[pHex, 16];
   {Px, Py} = MultiplyBasePoint[{Gx, Gy}, a, p, k];
   {ToHex[Px, 64], ToHex[Py, 64]}
  ];
```

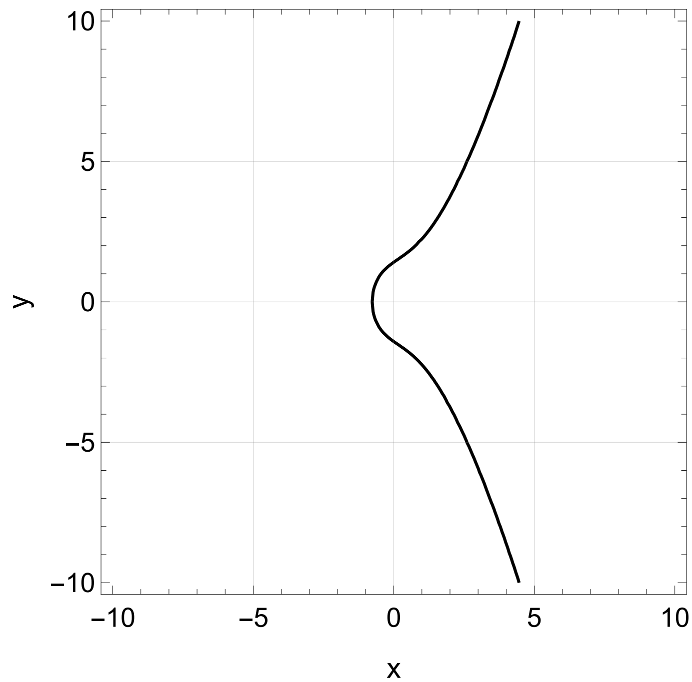
Example operations

```
In[38]:= k = 5;
         a = 2;
         p = 17;

In[41]:= G = {5, 1};

In[43]:= G2 = EllipticCurveDouble[G, a, p]

Out[43]= {6, 3}

In[44]:= G3 = EllipticCurveAdd[G, G2, p]

Out[44]= {10, 6}

In[45]:= MultiplyBasePoint[G, a, p, 10]

Out[45]= {7, 11}
```

Curves over the Reals Field

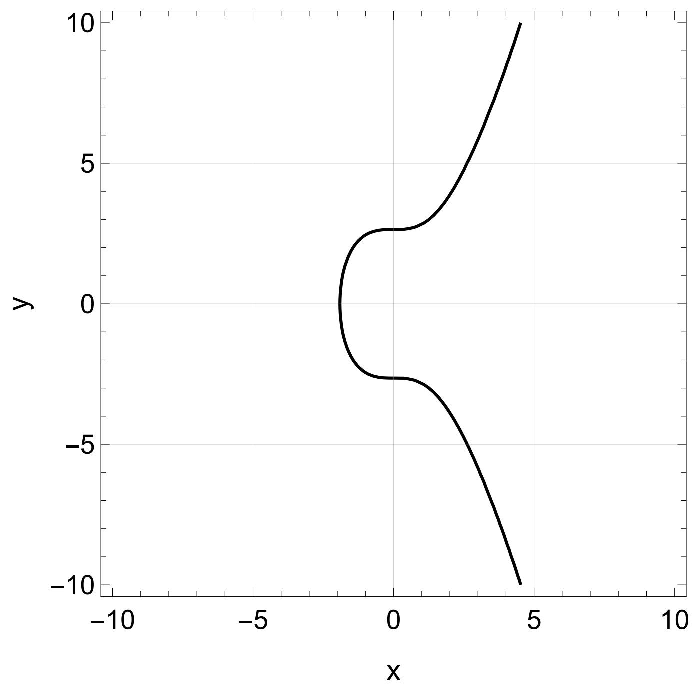
In[∘]:=
```
a = 2;
b = 2;
ContourPlot[y^2 == x^3 + a x + b, {x, -10, 10}, {y, -10, 10},
 PlotTheme → "Monochrome",
 Frame → True,
 BaseStyle → FontSize → 14,
 GridLines → Automatic,
 FrameLabel → {Style["x", 15], Style["y", 15]}
]
```

Out[∘]=



In[205]:=
```
a = 0;
b = 7;
ContourPlot[y^2 == x^3 + a x + b, {x, -10, 10}, {y, -10, 10},
 PlotTheme → "Monochrome",
 Frame → True,
 BaseStyle → FontSize → 14,
 GridLines → Automatic,
 FrameLabel → {Style["x", 15], Style["y", 15]}
]
```

Out[207]=



In[219]:=
```
G = {2, N@EllipticCurveEvaluate[2, a, b]}
```

Out[219]= {2, 3.87298}

In[220]:=
```
G2 = EllipticCurveDouble[G, a]
```

Out[220]= {-1.6, 1.70411}

In[226]:=
```
G3 = EllipticCurveAdd[G, G2]
```

Out[226]= {-0.037037, -2.64574}

In[227]:=
```
G4 = EllipticCurveAdd[G, G3]
```

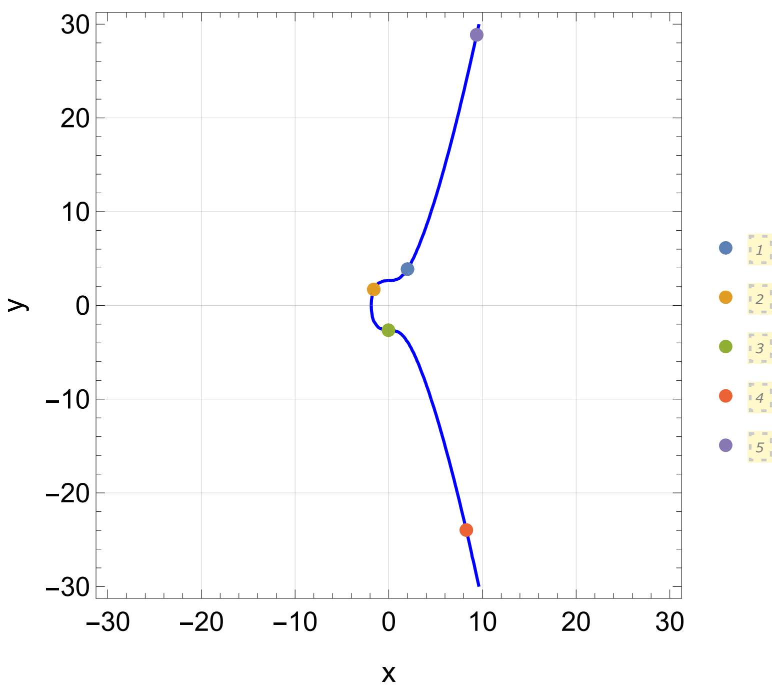
Out[227]= {8.27769, -23.9622}

In[228]:=
```
G5 = EllipticCurveAdd[G, G4]
```

Out[228]= {9.38258, 28.8613}

In[235]:=
```
Show[
 ContourPlot[y^2 == x^3 + a x + b, {x, -30, 30}, {y, -30, 30},
  PlotTheme → "Monochrome",
  Frame → True,
  BaseStyle → FontSize → 14,
  GridLines → Automatic,
  FrameLabel → {Style["x", 15], Style["y", 15]},
  ContourStyle → Blue
 ],
 ListPlot[{{G}, {G2}, {G3}, {G4}, {G5}},
  PlotStyle → PointSize[Large],
  PlotLegends → Automatic
 ]
]
```

Out[235]=



Curves over Finite Field

In[89]:=
```
CalculateCurvePoints[a_, b_, p_] := Flatten[ProgressTable[
    If[EllipticCurvePointQ[{x, y}, a, b, p], {x, y}, Nothing], {x, 0, p}, {y, 0, p}], 1];
```

$\mathbb{F}_{17}$

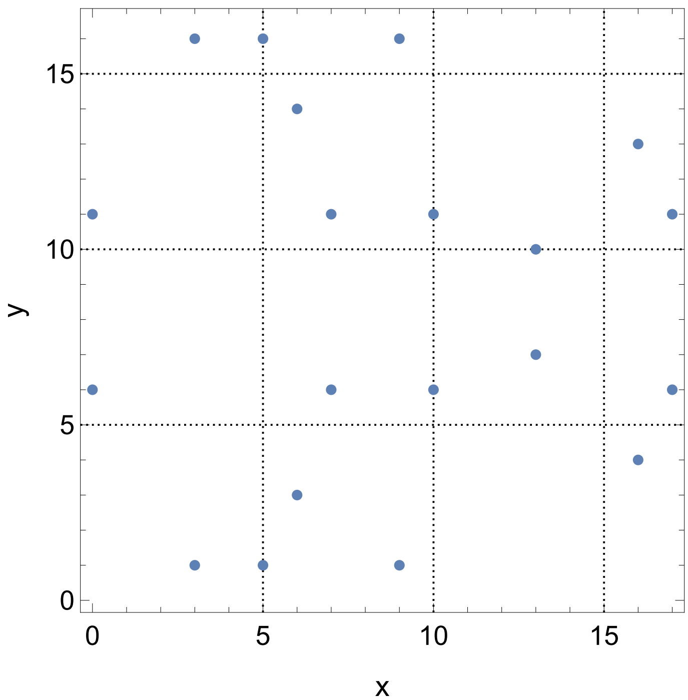
In[99]:=
```
k = 5;
a = 2;
b = 2;
p = 17;
G = {5, 1};
```

In[105]:=
```
GGroup = ComputeCyclicGroup[G, a, p];
Thread[Range[Length[GGroup]] → GGroup]
```

Out[106]=
```
{1 → {5, 1}, 2 → {6, 3}, 3 → {10, 6}, 4 → {3, 1}, 5 → {9, 16}, 6 → {16, 13}, 7 → {0, 6},
 8 → {13, 7}, 9 → {7, 6}, 10 → {7, 11}, 11 → {13, 10}, 12 → {0, 11}, 13 → {16, 4},
 14 → {9, 1}, 15 → {3, 16}, 16 → {10, 11}, 17 → {6, 14}, 18 → {5, 16}, 19 → ∞}
```

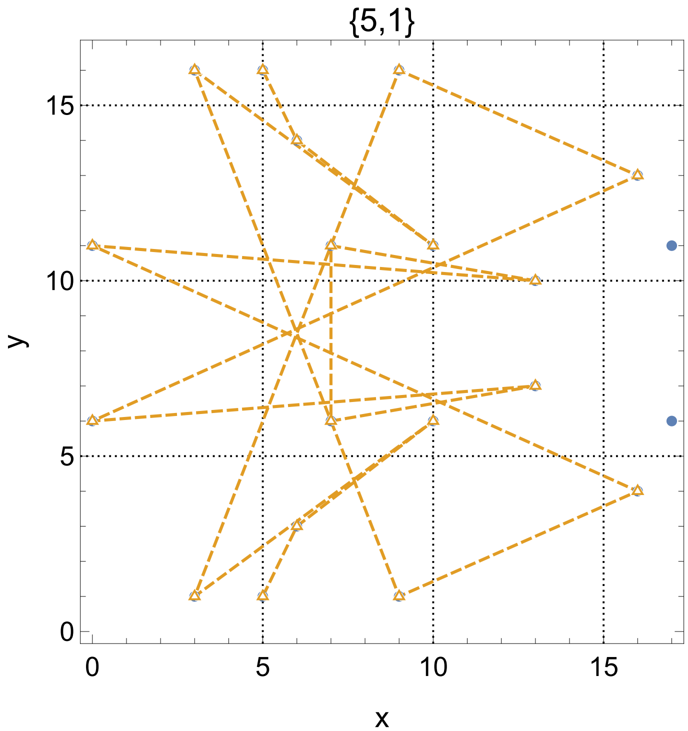
In[109]:=
```
curveSet = CalculateCurvePoints[a, b, p];
ListPlot[curveSet,
 AspectRatio → 1,
 PlotTheme → "Monochrome",
 Frame → True,
 BaseStyle → FontSize → 14,
 GridLines → Automatic,
 FrameLabel → {Style["x", 15], Style["y", 15]}
]
```

Out[110]=



In[111]:=
```
ListPlot[{curveSet, GGroup},
 AspectRatio → 1,
 Joined → {False, True},
 PlotTheme → "Monochrome",
 Frame → True,
 BaseStyle → FontSize → 14,
 GridLines → Automatic,
 FrameLabel → {Style["x", 15], Style["y", 15]},
 PlotLabel → "{5,1}"
]
```

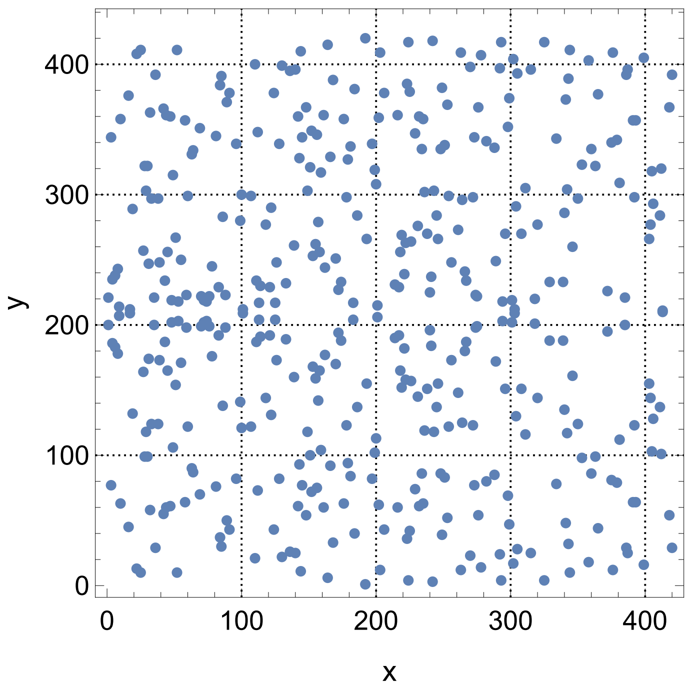
Out[111]=



$\mathbb{F}_{421}$

In[112]:=
```
p = 421;
curveSet = CalculateCurvePoints[a, b, p];
ListPlot[curveSet,
 AspectRatio → 1,
 PlotTheme → "Monochrome",
 Frame → True,
 BaseStyle → FontSize → 14,
 GridLines → Automatic,
 FrameLabel → {Style["x", 15], Style["y", 15]}
]
```

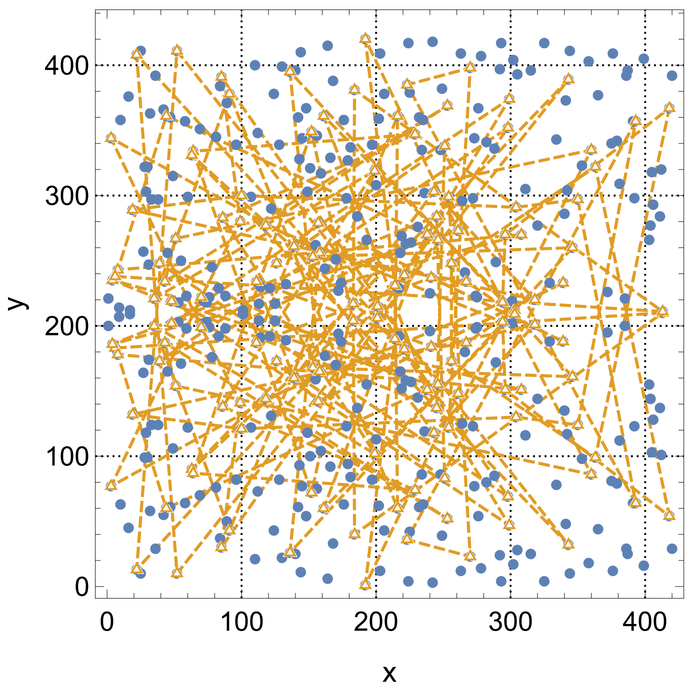
Out[114]=



In[53]:=
```
GGroup = ComputeCyclicGroup[{6, 183}, a, p];
```

In[54]:=
```
ListPlot[{curveSet, GGroup},
 AspectRatio → 1,
 Joined → {False, True},
 PlotTheme → "Monochrome",
 Frame → True,
 BaseStyle → FontSize → 14,
 GridLines → Automatic,
 FrameLabel → {Style["x", 15], Style["y", 15]}
]
```

Out[54]=



ECDSA signature and verification

Desciption:

ECDSA Sign

The ECDSA signing algorithm (RFC 6979) takes as input a message ***msg*** + a private key ***privKey*** and produces as output a **signature**, which consists of pair of integers {***r***, ***s***}. The **ECDSA signing** algorithm is based on the ElGamal signature scheme and works as follows (with minor simplifications):

1. Calculate the message **hash**, using a cryptographic hash function like SHA-256: ***h*** = hash(***msg***)
2. Generate securely a **random** number ***k*** in the range [1..***n***-1]
   - In case of **deterministic-ECDSA**, the value ***k*** is HMAC-derived from ***h*** + ***privKey*** (see RFC 6979)
3. Calculate the random point ***R*** = ***k*** * **G** and take its x-coordinate: ***r*** = ***R*.x**
4. Calculate the signature proof: ***s*** = $k^{-1} * (h + r * privKey) \pmod n$
   - The modular inverse $k^{-1} \pmod n$ is an integer, such that $k * k^{-1} \equiv 1 \pmod n$
5. Return the **signature** {***r***, ***s***}.

The calculated **signature** {***r***, ***s***} is a pair of integers, each in the range [1...***n***-1]. It encodes the random point ***R*** = ***k*** * **G**, along with a proof ***s***, confirming that the signer knows the message ***h*** and the private key ***privKey***. The proof ***s*** is by idea verifiable using the corresponding ***pubKey***.

ECDSA signatures are **2 times longer** than the signer's **private key** for the curve used during the signing process. For example, for 256-bit elliptic curves (like `secp256k1`) the ECDSA signature is 512 bits (64 bytes) and for 521-bit curves (like `secp521r1`) the signature is 1042 bits.

ECDSA Verify Signature

The algorithm to **verify a ECDSA signature** takes as input the signed message ***msg*** + the signature {***r***, ***s***} produced from the signing algorithm + the public key ***pubKey***, corresponding to the signer's private key. The output is boolean value: ***valid*** or ***invalid*** signature. The **ECDSA signature verify** algorithm works as follows (with minor simplifications):

1. Calculate the message **hash**, with the same cryptographic hash function used during the signing: ***h*** = hash(***msg***)
2. Calculate the modular inverse of the signature proof: ***s1*** = $s^{-1} \pmod n$
3. Recover the random point used during the signing: ***R'*** = (***h*** * **s1**) * **G** + (***r*** * ***s1***) * ***pubKey***
4. Take from ***R'*** its x-coordinate: ***r'*** = ***R'*.x**
5. Calculate the signature validation **result** by comparing whether ***r'*** == ***r***

The general idea of the signature verification is to **recover the point *R'*** using the public key and check whether it is same point ***R***, generated randomly during the signing process.

How Does it Work?

The **ECDSA signature** {***r***, ***s***} has the following simple explanation:

- The signing **signing** encodes a random point ***R*** (represented by its x-coordinate only) through elliptic-curve transformations using the private key ***privKey*** and the message hash ***h*** into a number ***s***, which is the **proof** that the message signer knows the private key ***privKey***. The signature {***r***, ***s***} cannot reveal the private key due to the difficulty of the **ECDLP problem**.
- The **signature verification** decodes the proof number ***s*** from the signature back to its original point ***R***, using the public key ***pubKey*** and the message hash ***h*** and compares the x-coordinate of the recovered ***R*** with the ***r*** value from the signature.

The Math behind the ECDSA Sign / Verify

Read this section **only if you like math**. Most developer may skip it.

How does the above sign / verify scheme work? It is not obvious, but let's play a bit with the equations.

The equation behind the recovering of the point ***R'***, calculated during the **signature verification**, can be transformed by replacing the ***pubKey*** with ***privKey*** * **G** as follows:

R' = (***h*** * ***s1***) * **G** + (***r*** * ***s1***) * ***pubKey*** =
= (***h*** * ***s1***) * **G** + (***r*** * ***s1***) * ***privKey*** * **G** =
= (***h*** + ***r*** * ***privKey***) * ***s1*** * **G**

If we take the number ***s*** = $k^{-1} * (h + r * privKey) \pmod n$, calculated during the signing process, we can calculate ***s1*** = $s^{-1} \pmod n$ like this:

s1 = $s^{-1} \pmod n$ =
= $(k^{-1} * (h + r * privKey))^{-1} \pmod n$ =
= $k * (h + r * privKey)^{-1} \pmod n$

Now, replace ***s1*** in the point ***R'***.

R' = (***h*** + ***r*** * ***privKey***) * ***s1*** * **G** =
= $(h + r * privKey) * k * (h + r * privKey)^{-1} \pmod n$ * **G** =
= **k** * **G**

The final step is to **compare** the **point** ***R'*** (decoded by the ***pubKey***) with the **point** ***R*** (encoded by the ***privKey***). The algorithm in fact compares only the x-coordinates of ***R'*** and ***R***: the integers ***r'*** and ***r***.

It is expected that ***r'*** == ***r*** if the signature is **valid** and ***r'*** ≠ ***r*** if the signature or the message or the public key is incorrect.

Implementation:

In[11]:=

```
ECDSASign[G_, a_, p_, m_, k_, hash_, privateKey_] := Block[{x, y, r, s},
   {x, y} = MultiplyBasePoint[G, a, p, k];
   r = x;
   s = Mod[ModularInverse[k, m] * (hash + r * privateKey), m];
   {r, s}
  ];
ECDSASignHex[{hexGx_, hexGy_}, aHex_, pHex_, mHex_, kHex_, hashHex_, privateKeyHex_] :=
  Block[{Gx, Gy, a, p, m, k, hash, privateKey, r, s},
   Gx = FromDigits[hexGx, 16];
   Gy = FromDigits[hexGy, 16];
   a = FromDigits[aHex, 16];
   p = FromDigits[pHex, 16];
   m = FromDigits[mHex, 16];
   k = FromDigits[kHex, 16];
   hash = FromDigits[hashHex, 16];
   privateKey = FromDigits[privateKeyHex, 16];
   {r, s} = ECDSASign[{Gx, Gy}, a, p, m, k, hash, privateKey];

   ToHex[r, 64] <> ToHex[s, 64]
  ];
ECDSAValidate[G_, a_, p_, m_, {r_, s_}, hash_, publicKey_] := Block[{w, u1, u2, X},
   If[r < 1 || r > m - 1, Return[False]];
   If[s < 1 || s > m - 1, Return[False]];

   w = ModularInverse[s, m];
   u1 = Mod[hash * w, m];
   u2 = Mod[r * w, m];
   X = EllipticCurveAdd[
     MultiplyBasePoint[G, a, p, u1], MultiplyBasePoint[publicKey, a, p, u2], p];
   Mod[First[X], m] == r
  ];
ECDSAValidateHex[{hexGx_, hexGy_}, aHex_, pHex_,
   mHex_, signature_, hashHex_, {publicKeyXHex_, publicKeyYHex_}] :=
  Block[{rHex, sHex, Gx, Gy, a, p, m, r, s, hash, publicKeyX, publicKeyY},
   {rHex, sHex} = StringPartition[signature, 64];
   Gx = FromDigits[hexGx, 16];
   Gy = FromDigits[hexGy, 16];
   a = FromDigits[aHex, 16];
   p = FromDigits[pHex, 16];
   m = FromDigits[mHex, 16];
   r = FromDigits[rHex, 16];
   s = FromDigits[sHex, 16];
   hash = FromDigits[hashHex, 16];
   publicKeyX = FromDigits[publicKeyXHex, 16];
   publicKeyY = FromDigits[publicKeyYHex, 16];

   ECDSAValidate[{Gx, Gy}, a, p, m, {r, s}, hash, {publicKeyX, publicKeyY}]
  ];
```

Example:

In[115]:=
```
a = 1;
b = 4;
p = 23;
G = {0, 2};
m = Length[ComputeCyclicGroup[G, a, p]];
```

In[120]:=
```
privateKey = 5
```

Out[120]= 5

In[121]:=
```
publicKey = MultiplyBasePoint[G, a, p, privateKey]
```

Out[121]= {7, 20}

In[ ]:=
```
messageHash = 17;
```

In[ ]:=
```
{r, s} = ECDSASign[G, a, p, m, k, messageHash, privateKey]
```

Out[ ]= {18, 11}

In[ ]:=
```
ECDSAValidate[G, a, p, m, {r, s}, messageHash, publicKey]
```

Out[ ]= True

Visualization:

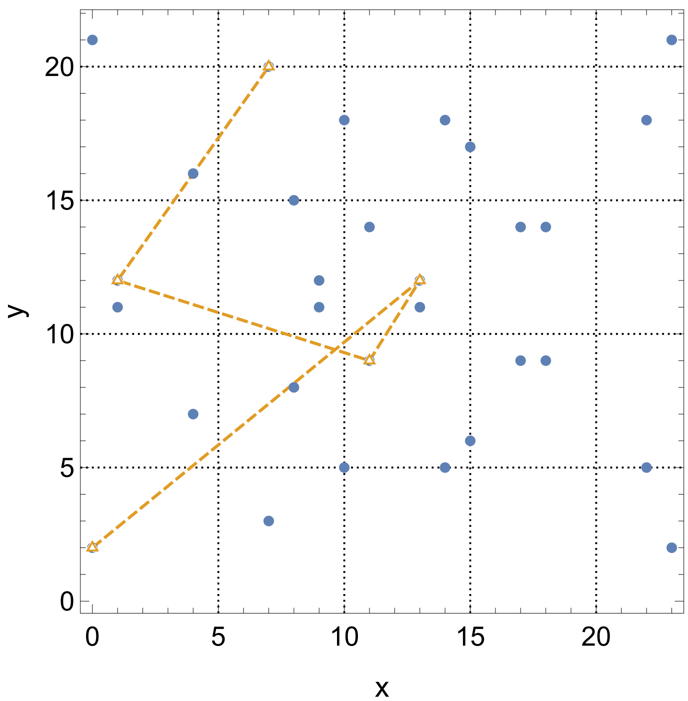
In[125]:=
```
path = MultiplicationPath[G, a, p, privateKey]
```

Out[125]= {{0, 2}, {13, 12}, {11, 9}, {1, 12}, {7, 20}}

In[123]:=
```
curveSet = CalculateCurvePoints[a, b, p];
```

In[127]:=
```
ListPlot[{curveSet, path},
 AspectRatio → 1,
 Joined → {False, True},
 PlotTheme → "Monochrome",
 Frame → True,
 BaseStyle → FontSize → 14,
 GridLines → Automatic,
 FrameLabel → {Style["x", 15], Style["y", 15]}
]
```

Out[127]=



Network simulation

secp256k1 parameters

In[139]:=
```
p = "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F";
m = "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141";
a = "0000000000000000000000000000000000000000000000000000000000000000";
b = "0000000000000000000000000000000000000000000000000000000000000007";
G = {"79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798",
   "483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8"};
```

Key Generation

The **ECDSA key-pair** consists of:

- **private key** (integer): ***privKey***
- **public key** (EC point): ***pubKey*** = ***privKey*** * **G**

The **private key** is generated as a **random integer** in the range [0...***n***-1]. The public key ***pubKey*** is a point on the elliptic curve, calculated by the EC point multiplication: ***pubKey*** = ***privKey*** * **G** (the private key, multiplied by the generator point **G**).

In[15]:=

```
ToHex[n_, pad_ : 64] := IntegerString[n, 16, pad];
Concatenate[l_] := Apply[Join, l];
GenerateNodes[G_, a_, m_, n_] :=
  Block[{mdec, privateKeys, publicKeys, privateKeysPadded, publicKeysPadded},
   mdec = FromDigits[m, 16];
   privateKeys = Table[ToHex[RandomInteger[{2, mdec - 1}], 64], n];
   publicKeys = Map[StringJoin[MultiplyBasePointHex[G, a, m, #]] &, privateKeys];

   MapThread[<|"PublicKey" → #1, "PrivateKey" → #2|> &, {publicKeys, privateKeys}]
  ];
```

```
In[145]:= networkNodes = GenerateNodes[G, a, p, 10]

Out[145]= {⟨|PublicKey →
    936864b882a618a25708fd32e3287cac10d289a27b22cd130d3042e8e25e05cf74ea37cdd13550a4c2ce4⋱
       2c6d0e222d3410c4a00ddd932261e8e3b8ccd65a68f, PrivateKey →
    39629ecaccd9f717c1f515f7415f5d13ca2468be1de6c6966ce21f3af4c8029f|⟩, ⟨|PublicKey →
    61249a16b3f56e3db642e5ca162beb0b64c3bee9e8768902c1482f6580e25de7a26590bb8cb37bbb4eb1f⋱
       dc9a49d3e645a5ca480101030c732609033ccbc2b8b, PrivateKey →
    1946334d1e5cd7bb60a83ee3b6170eab392be71a862d56e5c7ef72a4db647101|⟩, ⟨|PublicKey →
    5e6ac138d7041676c8cceaf562f5de44071aeddf90cf18e393b4288efdb8bb08048a73fee8baf07e577fd⋱
       2d99588749360dd0c46ad953967cf6775232a0ea8be, PrivateKey →
    05dd2f218f631dafc92c8e8042501a9084de22a8495030f32012c695ee78fe76|⟩, ⟨|PublicKey →
    8eb2357242224513080b264505b3a68d4db1dc36e1cfe0a50f0bd90a03f81de9376940dac7b6e6092ce7d⋱
       54250b70e6b5282f605a9a81803005f2703c8734ed2, PrivateKey →
    401d188c38232c4612c94312684d1b5627ab22bd2f29279f86565d41c00608a0|⟩, ⟨|PublicKey →
    8fdefc4cf847de6659a79c1641fa801aef8310b7b493cd5c810b4c3b5d58a99f43a298d156d9b71ee7b45⋱
       84c5c79e89dace77d4b96d1e6634c9fcdcd289b2b0c, PrivateKey →
    9fda40e27f6d01d9b428d9467621ee76970478c4582fc27d0ed89561a0ff82cd|⟩, ⟨|PublicKey →
    dded0656896dbaef475215738f5120b0f477c8f3110a65f5b30dedcb684ce600987642476c187fb88bef7⋱
       02061cb2a5aa78426f92397d4b59f290de923dd1ddc, PrivateKey →
    bf5fda59b79d949beb61d292235de6288b733bd2bd6b0856e76e992e24568b3f|⟩, ⟨|PublicKey →
    24203ad958810688c6690f64b2ab410aa246b2fc00cd26565ac503604562e87173227191b2b54eb2abf86⋱
       bd0711e77ffb74f7d371531f8b121e54e7ad3591569, PrivateKey →
    be8e3c235f19007081120a6f697ca8db0fb6eeb1e85996c775b797f6a784f7df|⟩, ⟨|PublicKey →
    9bcfa371088022e83287d55892844fb08284bb5e402123e1bc608c7ab4dbf664cab4065c5bec6a127fb6c⋱
       e11c3a8424f038aeff5a799a6b725362eb23e2459d2, PrivateKey →
    63a9b4f9f1c7550b233d9734aa38a7c3000f0970cf38b49820698bdcccc4330a|⟩, ⟨|PublicKey →
    94698ebc7e686035d1cf40c503e15622bb4a0d9c82d7a2e40a09e3a5e69327a9e8fd48281d666dcde3165⋱
       2a9392fae483b02b620ae9ea634f8151bbb3428f42b, PrivateKey →
    f983987749daa34f40f20be9e8c30dd58916760ee8eab3934589f1389d326ee0|⟩, ⟨|PublicKey →
    6798e006757a961c3394d90c1b1ab8fedce2ab563b68d8d17f65a729fa9aae25433e4d34f661f30e5206a⋱
       f049e154aa6018470f707969dba926605cc14c5e4bd,
  PrivateKey → c91bf857ddc61c2fdb4c1934e2a9ef4797f652f907a3b79ea4da204b935575fd|⟩}
```

In[133]:=
```
Transaction[sender_, receiver_, units_] := <|
   "Sender" → sender, "Receiver" → receiver, "Units" → units|>;
SimulateTransactions[networkNodes_, n_] := Block[{transactors},
   transactors =
    Take[Partition[RandomSample[networkNodes[[All, "PublicKey"]]], 2], n];
   Map[Transaction[First[#], Last[#], ToHex[RandomInteger[{1, 255}], 2]] &,
    transactors]
  ];

TransactionBlock[transactions_, prevBlockSignature_] :=
  prevBlockSignature <> StringJoin[Concatenate[Map[Values, transactions]]];
RandomNonceHex[m_] := IntegerString[RandomInteger[{1, FromDigits[m, 16]}], 16];

MineBlock[G_, a_, p_, m_, hash_, privateKey_, difficulty_ : 2] :=
  Block[{i, signature, leadingDigits},
   i = 1;
   signature = "11";
   leadingDigits = StringJoin[ConstantArray["0", difficulty]];

   While[StringTake[signature, 2] ≠ leadingDigits,
    signature = ECDSASignHex[G, a, p, m, RandomNonceHex[m], hash, privateKey];
    i++;
   ];
   {i, signature}
  ];
```

In[146]:=
```
transactions = SimulateTransactions[networkNodes, 3]
```

Out[146]=
```
{<|Sender →
  94698ebc7e686035d1cf40c503e15622bb4a0d9c82d7a2e40a09e3a5e69327a9e8fd48281d666dcde3165⋱
     2a9392fae483b02b620ae9ea634f8151bbb3428f42b, Receiver →
  8eb2357242224513080b264505b3a68d4db1dc36e1cfe0a50f0bd90a03f81de9376940dac7b6e6092ce7d⋱
     54250b70e6b5282f605a9a81803005f2703c8734ed2, Units → 6d|>, <|Sender →
  5e6ac138d7041676c8cceaf562f5de44071aeddf90cf18e393b4288efdb8bb08048a73fee8baf07e577fd⋱
     2d99588749360dd0c46ad953967cf6775232a0ea8be, Receiver →
  dded0656896dbaef475215738f5120b0f477c8f3110a65f5b30dedcb684ce600987642476c187fb88bef7⋱
     02061cb2a5aa78426f92397d4b59f290de923dd1ddc, Units → e0|>, <|Sender →
  9bcfa371088022e83287d55892844fb08284bb5e402123e1bc608c7ab4dbf664cab4065c5bec6a127fb6c⋱
     e11c3a8424f038aeff5a799a6b725362eb23e2459d2, Receiver →
  61249a16b3f56e3db642e5ca162beb0b64c3bee9e8768902c1482f6580e25de7a26590bb8cb37bbb4eb1f⋱
     dc9a49d3e645a5ca480101030c732609033ccbc2b8b, Units → 0d|>}
```

In[151]:=
```
block = TransactionBlock[transactions, ToHex[0, 128]]
```

Out[151]=
```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    00000000000000000000000000000000000000009469
    8ebc7e686035d1cf40c503e15622bb4a0d9c82d7a2e40a09e3a5e69327a9e8fd48281d666dcde31652a939
    2fae483b02b620ae9ea634f8151bbb3428f42b8eb2357242224513080b264505b3a68d4db1dc36e1cfe0a
    50f0bd90a03f81de9376940dac7b6e6092ce7d54250b70e6b5282f605a9a81803005f2703c8734ed26d5e
    6ac138d7041676c8cceaf562f5de44071aeddf90cf18e393b4288efdb8bb08048a73fee8baf07e577fd2d
    99588749360dd0c46ad953967cf6775232a0ea8bedded0656896dbaef475215738f5120b0f477c8f3110a
    65f5b30dedcb684ce600987642476c187fb88bef702061cb2a5aa78426f92397d4b59f290de923dd1ddce
    09bcfa371088022e83287d55892844fb08284bb5e402123e1bc608c7ab4dbf664cab4065c5bec6a127fb6
    ce11c3a8424f038aeff5a799a6b725362eb23e2459d261249a16b3f56e3db642e5ca162beb0b64c3bee9e
    8768902c1482f6580e25de7a26590bb8cb37bbb4eb1fdc9a49d3e645a5ca480101030c732609033ccbc2b
    8b0d
```

In[152]:=
```
hash = Hash[block, "SHA256", "HexString"]
```

Out[152]=
```
2d61c71a084e1f019be8695470bd70e17da372ae80df9b3ffc247560923c7531
```

Simulation of nodes

In[158]:=
```
AddRewardToBlock[transactions_, publicKey_] := Append[
   transactions, <|"Sender" → ToHex[0, 128], "Receiver" → publicKey, "Units" → "01"|>];
SimulateNode[prevBlockSignature_, transactions_, privateKey_, publicKey_,
   {G_, a_, p_, m_}] := Block[{transactionsWithReward, block, hash, publicKeyParts},
   transactionsWithReward = AddRewardToBlock[RandomSample[transactions], publicKey];
   block = TransactionBlock[transactionsWithReward, prevBlockSignature];
   hash = Hash[block, "SHA256", "HexString"];
   publicKeyParts = StringPartition[publicKey, 64];

   {block, publicKeyParts, MineBlock[G, a, p, m, hash, privateKey, 2]}
  ];
```

In[165]:=
```
nodesSimulation = ProgressMap[SimulateNode[ToHex[0, 128], transactions,
     #["PrivateKey"], #["PublicKey"], {G, a, p, m}] &, networkNodes];
```

In[166]:=
```
firstMine = First[MinimalBy[nodesSimulation[[All, 3]], First]];
```

In[167]:=
```
firstMiner  = FirstCase[nodesSimulation, {_, _, firstMine}];
```

```
In[168]:= newBlock = <|"TransactionBlock" → First[firstMiner],
             "Signature" -> Last[Last[firstMiner]], "PublicKey" → firstMiner[[2]]|>

Out[168]= <|TransactionBlock →
            00000000000000000000000000000000000000000000000000000000000000000000000000000000000000
             000000000000000000000000000000000000000000
             9bcfa371088022e83287d55892844fb08284bb5e402123e1bc608c7ab4dbf664cab4065c5bec6a127f
             b6ce11c3a8424f038aeff5a799a6b725362eb23e2459d261249a16b3f56e3db642e5ca162beb0b64c
             3bee9e8768902c1482f6580e25de7a26590bb8cb37bbb4eb1fdc9a49d3e645a5ca480101030c73260
             9033ccbc2b8b0d5e6ac138d7041676c8cceaf562f5de44071aeddf90cf18e393b4288efdb8bb08048
             a73fee8baf07e577fd2d99588749360dd0c46ad953967cf6775232a0ea8bedded0656896dbaef4752
             15738f5120b0f477c8f3110a65f5b30dedcb684ce600987642476c187fb88bef702061cb2a5aa7842
             6f92397d4b59f290de923dd1ddce094698ebc7e686035d1cf40c503e15622bb4a0d9c82d7a2e40a09
             e3a5e69327a9e8fd48281d666dcde31652a9392fae483b02b620ae9ea634f8151bbb3428f42b8eb23
             57242224513080b264505b3a68d4db1dc36e1cfe0a50f0bd90a03f81de9376940dac7b6e6092ce7d5
             4250b70e6b5282f605a9a81803005f2703c8734ed26d0000000000000000000000000000000000000
             000000000000000000000000000000000000000000000000000000000000000000000000000000000
             0000000008eb2357242224513080b264505b3a68d4db1dc36e1cfe0a50f0bd90a03f81de9376940d
             ac7b6e6092ce7d54250b70e6b5282f605a9a81803005f2703c8734ed201, Signature →
            007d7f5ee6e52b4f41d9d5229ec38ddf971db0af00a3c9c46c3115a90378f60ee9d2ee31e60358967b6fa2
             1a19f7069a5b28dec501d1fc42a7cdf69d98f770b9,
           PublicKey → {8eb2357242224513080b264505b3a68d4db1dc36e1cfe0a50f0bd90a03f81de9,
             376940dac7b6e6092ce7d54250b70e6b5282f605a9a81803005f2703c8734ed2}|>

In[176]:= hash = Hash[newBlock["TransactionBlock"], "SHA256", "HexString"];
          signature = newBlock["Signature"];
          publicKey = newBlock["PublicKey"];

In[179]:= ECDSAValidateHex[G, a, p, m, signature, hash, publicKey]

Out[179]= True
```

Iteration 2

```
In[180]:= transactions = SimulateTransactions[networkNodes, 3]

Out[180]= {<|Sender →
             8fdefc4cf847de6659a79c1641fa801aef8310b7b493cd5c810b4c3b5d58a99f43a298d156d9b71ee7b45
               84c5c79e89dace77d4b96d1e6634c9fcdcd289b2b0c, Receiver →
             936864b882a618a25708fd32e3287cac10d289a27b22cd130d3042e8e25e05cf74ea37cdd13550a4c2ce4
               2c6d0e222d3410c4a00ddd932261e8e3b8ccd65a68f, Units → ee|>, <|Sender →
             6798e006757a961c3394d90c1b1ab8fedce2ab563b68d8d17f65a729fa9aae25433e4d34f661f30e5206a
               f049e154aa6018470f707969dba926605cc14c5e4bd, Receiver →
             61249a16b3f56e3db642e5ca162beb0b64c3bee9e8768902c1482f6580e25de7a26590bb8cb37bbb4eb1f
               dc9a49d3e645a5ca480101030c732609033ccbc2b8b, Units → 69|>, <|Sender →
             dded0656896dbaef475215738f5120b0f477c8f3110a65f5b30dedcb684ce600987642476c187fb88bef7
               02061cb2a5aa78426f92397d4b59f290de923dd1ddc, Receiver →
             5e6ac138d7041676c8cceaf562f5de44071aeddf90cf18e393b4288efdb8bb08048a73fee8baf07e577fd
               2d99588749360dd0c46ad953967cf6775232a0ea8be, Units → 3a|>}

In[182]:= prevBlockSignature = newBlock["Signature"];
```

In[183]:=
```
nodesSimulation = ProgressMap[SimulateNode[prevBlockSignature,
    transactions, #["PrivateKey"], #["PublicKey"], {G, a, p, m}] &, networkNodes];
```

In[184]:=
```
firstMine = First[MinimalBy[nodesSimulation[[All, 3]], First]];
firstMiner  = FirstCase[nodesSimulation, {_, _, firstMine}];
newBlock2 = <|"TransactionBlock" → First[firstMiner],
  "Signature" -> Last[Last[firstMiner]], "PublicKey" → firstMiner[[2]]|>
```

Out[186]=
```
<|TransactionBlock →
 007d7f5ee6e52b4f41d9d5229ec38ddf971db0af00a3c9c46c3115a90378f60ee9d2ee31e60358967b6fa2⋱
    1a19f7069a5b28dec501d1fc42a7cdf69d98f770b9dded0656896dbaef475215738f5120b0f477c8f⋱
    3110a65f5b30dedcb684ce600987642476c187fb88bef702061cb2a5aa78426f92397d4b59f290de9⋱
    23dd1ddc5e6ac138d7041676c8cceaf562f5de44071aeddf90cf18e393b4288efdb8bb08048a73fee⋱
    8baf07e577fd2d99588749360dd0c46ad953967cf6775232a0ea8be3a6798e006757a961c3394d90c⋱
    1b1ab8fedce2ab563b68d8d17f65a729fa9aae25433e4d34f661f30e5206af049e154aa6018470f70⋱
    7969dba926605cc14c5e4bd61249a16b3f56e3db642e5ca162beb0b64c3bee9e8768902c1482f6580⋱
    e25de7a26590bb8cb37bbb4eb1fdc9a49d3e645a5ca480101030c732609033ccbc2b8b698fdefc4cf⋱
    847de6659a79c1641fa801aef8310b7b493cd5c810b4c3b5d58a99f43a298d156d9b71ee7b4584c5c⋱
    79e89dace77d4b96d1e6634c9fcdcd289b2b0c936864b882a618a25708fd32e3287cac10d289a27b2⋱
    2cd130d3042e8e25e05cf74ea37cdd13550a4c2ce42c6d0e222d3410c4a00ddd932261e8e3b8ccd65⋱
    a68fee000000000000000000000000000000000000000000000000000000000000000000000000000⋱
    000000000000000000000000000000000000000000000000000008eb2357242224513080b264505b3⋱
    a68d4db1dc36e1cfe0a50f0bd90a03f81de9376940dac7b6e6092ce7d54250b70e6b5282f605a9a81⋱
    803005f2703c8734ed201, Signature →
 00ade813fa79a3f34805c227c02df4b7ce02e8854b64c084735a1073cde85f99542cbdc161e9e14305ec2e⋱
    172b7fbdde80e48fc9de6cfe62b0e98112972408cd,
PublicKey → {8eb2357242224513080b264505b3a68d4db1dc36e1cfe0a50f0bd90a03f81de9,
  376940dac7b6e6092ce7d54250b70e6b5282f605a9a81803005f2703c8734ed2}|>
```

Iteration 3

In[187]:=
```
transactions = SimulateTransactions[networkNodes, 3]
```

Out[187]=
```
{<|Sender →
  8fdefc4cf847de6659a79c1641fa801aef8310b7b493cd5c810b4c3b5d58a99f43a298d156d9b71ee7b45⋱
    84c5c79e89dace77d4b96d1e6634c9fcdcd289b2b0c, Receiver →
  61249a16b3f56e3db642e5ca162beb0b64c3bee9e8768902c1482f6580e25de7a26590bb8cb37bbb4eb1f⋱
    dc9a49d3e645a5ca480101030c732609033ccbc2b8b, Units → de|>, <|Sender →
  936864b882a618a25708fd32e3287cac10d289a27b22cd130d3042e8e25e05cf74ea37cdd13550a4c2ce4⋱
    2c6d0e222d3410c4a00ddd932261e8e3b8ccd65a68f, Receiver →
  24203ad958810688c6690f64b2ab410aa246b2fc00cd26565ac503604562e87173227191b2b54eb2abf86⋱
    bd0711e77ffb74f7d371531f8b121e54e7ad3591569, Units → 08|>, <|Sender →
  9bcfa371088022e83287d55892844fb08284bb5e402123e1bc608c7ab4dbf664cab4065c5bec6a127fb6c⋱
    e11c3a8424f038aeff5a799a6b725362eb23e2459d2, Receiver →
  8eb2357242224513080b264505b3a68d4db1dc36e1cfe0a50f0bd90a03f81de9376940dac7b6e6092ce7d⋱
    54250b70e6b5282f605a9a81803005f2703c8734ed2, Units → f0|>}
```

In[188]:=
```
prevBlockSignature = newBlock2["Signature"];
```

In[189]:=
```
nodesSimulation = ProgressMap[SimulateNode[prevBlockSignature,
    transactions, #["PrivateKey"], #["PublicKey"], {G, a, p, m}] &, networkNodes];
```

```
In[190]:= firstMine = First[MinimalBy[nodesSimulation[[All, 3]], First]];
firstMiner = FirstCase[nodesSimulation, {_, _, firstMine}];
newBlock3 = <|"TransactionBlock" → First[firstMiner],
  "Signature" -> Last[Last[firstMiner]], "PublicKey" → firstMiner[[2]]|>
```

```
Out[192]= <|TransactionBlock →
  00ade813fa79a3f34805c227c02df4b7ce02e8854b64c084735a1073cde85f99542cbdc161e9e14305ec2e
     172b7fbdde80e48fc9de6cfe62b0e98112972408cd8fdefc4cf847de6659a79c1641fa801aef8310b
     7b493cd5c810b4c3b5d58a99f43a298d156d9b71ee7b4584c5c79e89dace77d4b96d1e6634c9fcdcd
     289b2b0c61249a16b3f56e3db642e5ca162beb0b64c3bee9e8768902c1482f6580e25de7a26590bb8
     cb37bbb4eb1fdc9a49d3e645a5ca480101030c732609033ccbc2b8bde9bcfa371088022e83287d558
     92844fb08284bb5e402123e1bc608c7ab4dbf664cab4065c5bec6a127fb6ce11c3a8424f038aeff5a
     799a6b725362eb23e2459d28eb2357242224513080b264505b3a68d4db1dc36e1cfe0a50f0bd90a03
     f81de9376940dac7b6e6092ce7d54250b70e6b5282f605a9a81803005f2703c8734ed2f0936864b88
     2a618a25708fd32e3287cac10d289a27b22cd130d3042e8e25e05cf74ea37cdd13550a4c2ce42c6d0
     e222d3410c4a00ddd932261e8e3b8ccd65a68f24203ad958810688c6690f64b2ab410aa246b2fc00c
     d26565ac503604562e87173227191b2b54eb2abf86bd0711e77ffb74f7d371531f8b121e54e7ad359
     156908000000000000000000000000000000000000000000000000000000000000000000000000000
     0000000000000000000000000000000000000000000000dded0656896dbaef475215738f51
     20b0f477c8f3110a65f5b30dedcb684ce600987642476c187fb88bef702061cb2a5aa78426f92397d
     4b59f290de923dd1ddc01, Signature →
  008bc9d3eee592ea92a36266258f39cf9a56e8b42b803685c93432738bce647d6123d9eb5d39c24eae1b7b
     1877e859d39e22fbbef67e37a3289dad0c6d3a0c72,
 PublicKey → {dded0656896dbaef475215738f5120b0f477c8f3110a65f5b30dedcb684ce600,
   987642476c187fb88bef702061cb2a5aa78426f92397d4b59f290de923dd1ddc}|>
```

First 8 trials

```
In[201]:= hash = Hash[block, "SHA256", "HexString"]
```

```
Out[201]= a397a3076b6d92b723348b671be686c6d8fbcd19757ca8cda5fc272445895fad
```

```
In[ ]:= signatures = Table[ECDSASignHex[G, a, p, m, ToHex[k, 64], hash, privateKey], {k, 2, 10}]
```

```
Out[ ]= {c6047f9441ed7d6d3045406e95c07cd85c778e4b8cef3ca7abac09b95c709ee5530ba3f44d230db31a114b9
     fac405628476fc487ab8812324438d4933af238ea,
 f9308a019258c31049344f85f89d5229b531c845836f99b08601f113bce036f93850de47cb04a677d8db98a
     cc82ecbef9402650f7278813b4bd5c8e7cf45e3a8,
 e493dbf1c10d80f3581e4904930b1404cc6c13900ee0758474fa94abe8c4cd13d0add5c9d2a78639972ee30
     b87758c09adee21d48e2b425d6710acf835221d19,
 2f8bde4d1a07209355b4a7250a5c5128e88b84bddc619ab7cba8d569b240efe46d7b6e78ebeb0059caf45c4
     2358fa715c7cf400bd8cef5949b7f125da76ca298,
 fff97bd5755eeea420453a14355235d382f6472f8568a18b2f057a1460297556396502ce141a28bdfd8eb78
     3dfca183ba8e999f8a13cff93cdc651679c8b8b85,
 5cbdf0646e5db4eaa398f365f2ea7a0e3d419b7e0330e39ce92bddedcac4f9bc6ff0117e33c9a8ebc2af7f6
     81fb5e371f46a655d2441ab40078bacc9ac32e824,
 2f01e5e15cca351daff3843fb70f3c2f0a1bdd05e5af888a67784ef3e10a2a0125525e7381636ed50a5c555
     84595223489c34f469204dfa774c3652b7f2bd652,
 acd484e2f0c7f65309ad178a9f559abde09796974c57e714c35f110dfc27ccbe6d8b4aec4af971c5270fc15
     8da2ef468f696d3c1bd6cdf471a9373d795e56afd,
 a0434d9e47f3c86235477c7b1ae6ae5d3442d49b1943c2b752a68e2a47e247c7c26d422b5b092c50792f36e
     3cf15db61781483957a206f90179c09c5c5ce42ba}
```