IMPLEMENTATION DOCUMENT

BRENDA CHEN

generating BGS tree:

Algorithm 1 BGSTree(n): generate a BGS tree with depth n

Require: $n \geq 2$
Ensure: the output is a BGS-style markoff tree with depth n

```
Tree = MarkoffNode(1, 1, 1, 1)                    ▷ initialize root node of (1, 1, 1)
for i in range (N): do
    for child in children of Tree do
        grandChild1 = Rot1(child)
        grandChild2 = Rot2(child)
        grandChild3 = Rot3(child)
        child.children = removeduplicates[grandChild1, grandChild2, grandChild3]   ▷
    remove identical children add the children nodes to tree
    end
end
return Tree
```

Note: Other trees are generated in a similar algorithm with different formula to produce the children nodes.

Date: June 7, 2024.

set of functions that graph distribution of sizes at a given level
NOTE: input i specifies types of construction of the tree: BGS, Zagier, Involution

Algorithm 2 findSizes(N, i): return the sizes of nodes at level n

Require: $n \geq 1$
Ensure: the output is a list of sizes of all nodes at depth n

$Tree = MarkoffTree(n)$ ▷ generate a BGS style Markoff Tree of level n
$sizeList = [\ ]$
for *child in Tree.childNodes* **do**
 $sizeList+ = [max(chlid.Triple)]$
end
return sizeList

Algorithm 3 plotDistLog(N, i): graph histogram of log_3 sizes at a given level

Require: $N \geq 0,$
$Sizes = log_3 k for k in find Sizes(N, i)$
$graphHistogram(x = Sizes, y = Frequency)$

Algorithm 4 truncatePlotDist(N, i, Percentile): graph a percentile of sizes at level N

Require: $N \geq 0,\ i \geq 0,\ percentile \geq 0$
$Sizes = findPercentile(findSizes(N, i), percentile)$ ▷ find the truncated set of sizes
$graphHistogram(x = Sizes, y = Frequency)$

sample random points at a given level and graph sizes of a collection of random points

Algorithm 5 randomPathBGS(L, N): generate a set of N points at level L obtained by randomly choosing a rotation between $rot1$, $rot2$, and $rot3$ at each step

Require: $N \geq 0, L \geq 2$
Ensure: returns a list of positive integers

$startPoint = [1, 1, 2]$ ▷ we start at level 2 of Markoff Tree since the first two levels only have one distinct node
$returnList = []$
for *i in range(N)* **do**
 for *o in range(L-2)* **do**
  $startPoint = randomRotation(startPoint)$ ▷ apply random rotation
 end
 $returnList+ = [startPoint]$
 $startPoint = [1, 1, 2]$
end
$return returnList$

Algorithm 6 graphRandomBGS(L, N): graphs histogram of distribution of sizes of N randomly chosen points at level L of the BGS tree

Require: $N \geq 0, L \geq 2$
$List = randomPathBGD(L, N)$
$graph(xaxis = List, yaxis = frequency)$ ▷ graph the histogram w.r.t sizes of nodes

Set of Functions that Generates Markoff $\bmod p$ **tree** (G_p)

Algorithm 7 returnMod(List, p): given a List, returns the list with all elements mod p

Require: $p \geq 0$
$returnList = [element \mod p \text{ for } element\ in\ List]$
$return\ returnList$

The following two functions (combineNumOrd and combineUnord)(though combineUnord is slightly more integrated) represents different ways a vertex is identified in G_p, for example, whether we want to recognize $(1, 2, 3)$ and $(2, 3, 1)$ as different or the same vertex

Algorithm 8 combineNumOrd(a, b, c, p): given 3 numbers, return the combination of 3 numbers into one number in increasing order of the 3 inputs

$return\ (int)(str(a) \mod p +' 000' + str(b) \mod p +' 000' + str(c) \mod p)$

Algorithm 9 combineUnord(a, b, c, p): given 3 numbers, return the combination of 3 numbers $\mod p$ in the order in which a, b, c were given

Require: $N \geq 0, L \geq 2$
$Sorted = sort[a, b, c]$
$return\ (int)(str(Sorted[0]) +' 000' + str(Sorted[1]) +' 000' + str(Sorted[2]))$

Here, the function names a node $(x1, x2, x3)$ with a string $x_1000x_2000x_3000$; in other words, when p is large enough, there is no longer an injection between the set of all nodes and all strings in the form $x_1000x_2000x_3000$. This could be solved by increasing number of zeroes that separates the indices.

The following algorithms use various ways to implement/draw G_p

---

Algorithm 10 graphmodPLev(L, P): given level L and prime P, generate G_p up to level L of the BGS tree

---

Require: $L \geq 2, P > 0$
  $edgeList = [\ ]$
  $Tree = BGSTree(L)$
  for *node in PreOrderIteration(Tree)* **do**
    for *chlid in node.children* ▷ *iterate through children of node* **do**
      $markoffNode = combineNum(node.data1, node.data2, node, data3, P)$
      $markoffchild = combineNum(child.data1, child.data2, child.data3, P$
      $edgeList+ = [Markoffnode, Markoffchild]$ ▷ add edge from parent to child
    end
  end
  $graph(edges = edgeList)$ ▷ draw a graph such that the edges are in edgeList

---

Note: There is an implementation of unordered graphmod version which is almost identical as the above. It could be found in the code file.

Stopping at a specified level might not give us a complete G_p since some points might be at higher levels in the tree. The following two implementations uses properties of markoff mod p triples to obtain complete markoff mod p graphs.

Algorithm 11 graphmodP1 (p) : add edges to graph until number of nodes matches expected number of nodes

---

Require: $p > 0$
  if $p \mod 4 == 1$ **then**
    $expectedSize = p^2 + 3p$
  end
  if $p \mod 4 == 3$ **then**
    $expectedSize = p^2 - 3p$
  end
  $G = empty\ graph$
  $currentLevel = 0$
  while *currentSize* < *expectedSize* **do**
    $largeTree = BGSTree(currentLevel + 2)$
    $ParentList = nodes\ in\ LargeTree\ such\ that\ their\ level = currentLevel$
    $G.addedges(combineUnord(Parent\ in\ ParentList), combineUnord(child\ in\ Parent.chldren))$
  ▷ add edges to G
    $currentSize = G.nodesize$ ▷ update currentSize to current node size of G
    currentLevel+=1
  end
  draw G
  return $L + 1$ ▷ return the level where node adding stops

The following algorithm is a helper function to generate all Markoff triples.

Algorithm 12 checkMarkoff(List, p): check if a given List of 3 numbers is a Markoff mod p triple

Require: $Len(List) == 3, p > 0$
Ensure: *outputs an integer in*$\{0, 1\}$
  Let $List = [x_1, x_2, x + 3]$
  if $x_1^2 + x_2^2 + x_3^2 = 3x_1x_2x_3 \mod p$ **then**
    $return 1$
  end
  return 0

Algorithm 13 graphModp2(p): generate G_p by first generating all markoff mod p triples and connecting them to their children via applying rotations

Require: $p > 0$
  $allMarkoff = all\ Markoff\ Triples \mod p$
  $G = emptygraph$
  for *triple in allMarkoff* **do**
    $G.addedge(combineNumUnord(triple), Rot1(triple))$
    $G.addedge(combineNumUnord(triple), Rot2(triple))$
    $G.addedge(combineNumUnord(triple), Rot3(triple))$
  end
  $graph(G)$

The following algorithms are written to investigate properties of markoff trees with respect to different constructions: Zagier, BGS, involution. In particular, we try to investigate whether

Algorithm 14 verifyDetail(n, L): randomly select a set of n points at level L of the BGS tree, return information about whether the points are at level L or smaller levels of Zagier tree and involution tree

Require: $n > 0, L > 2$
  $BGSTree = BGSTree(L)$
  $ZagierNodes = ZagierTree(L).nodes$
  $InvTreeNodes = InvolutionTree(L).nodes$
  $ZagierNodesL = Nodes\ at\ level\ L\ in\ Zagier\ Tree$
  $InvNodesL = Nodes\ at\ level\ L\ in\ Involution\ Tree$
    ▷ initialize sets of all nodes and sets of nodes at level L for different trees
  $RandomBGSTriple = a\ randomly\ generated\ set\ of\ n\ points\ at\ level\ L\ of\ BGS\ Tree$
  $ZagierDifference = RandomBGSTriples/(RandomBGSTriple \cap ZagierNodes)$
  $InvDifference = RandomBGSTriples/(RandomBGSTriple \cap InvNodes)$
    ▷ check if the randomly selected BGS nodes appear in the level L zagier/inv tree
  $ZagierDifferenceL = RandomBGSTriples/(RandomBGSTriple \cap ZagierNodesL)$
  $InvDifferenceL = RandomBGSTriples/(RandomBGSTriple \cap InvNodesL)$
    ▷ check if the random set appears at level L if Zagier/inv trees
  return $[ZagierDifferenceL, InvDifferenceL, ZagierDifference, InvDifference]$

After investigating some properties of different variants of markoff graphs as well as G_p when the trees grows large, we start to implementing the path finding algorithm outlined in the paper by BGS.

checks if a point is in the cage

computing rotation order

The following algorithm finds rotational order by its definition: compute the i^th rotation and find k such that $(rot_i)^k List = List$

Algorithm 15 findOrder(List, i, p): given a triple, compute its i^th order with respect to prime p

---

Require: $length(List) == 3, i \in \{0, 1, 2\}, p > 0$
Initial = List
for k *in range* $p^2 + 1$ **do**
  $Initial = Rot_i(Initial)$
  if $Initial \mod p == List$ **then**
    $return\ k + 1$
  end
end

The following algorithm finds the i^th rotation order of triple (x_1, x_2, x_3) by computing order of matrix $[[0, 1], [-1, 3x_i]]$.

Algorithm 16 findOrderMat(List, i, p): return the i^{th} order of the triple (input as List) with respect to prime p

---

Require: $len(List) == 3, i \in \{0, 1, 2\}, p > 0$
$Matrix = \begin{bmatrix} 0 & 1 \\ -1 & 3x_i \end{bmatrix}$
$MatrixCopy = Matrix$
for k *in range* $p^2 + 1$ **do**
  $MatrixCopy = Matrix \cdot MatrixCopy$
  if $MatrixCopy == Matrix \mod p$ **then**
    $return\ k + 1$
  end
end

Functions below are mainly used to check if a point is in the cage or not.

Algorithm 17 checki (List, i, p): returns 1 if the i^th rotation order is p-1, p+1, p, or 2p, return 0 otherwise(checks if the ith coordinate is maximal)

Require: $len(List) == 3, i \in \{0, 1, 2\}, p > 0$
$O = findOrder(List, i, p)$
$D = findDiscriminant(List, i, p)$ ▷ find the i^th discriminant
if $D == 0$ *and (O == p or O == 2*p)* **then**
return 1
end
▷ parabolic
if *(O == p-1 or O == p+1) and (D != 0)* **then**
return 1
end
▷ elliptic or hyperbolic

Algorithm 18 checkSqrtp(List, p): checks if the i^th order $\geq \sqrt{p}, i \in \{1, 2, 3\}$

Require: $len(List) == 3, p > 0$
if $findOrder(List, i, p) \geq \sqrt{p}, i \in \{0, 1, 2\}$ **then**
return 1
end

note: the following findMaxOrd doesn't actually do what we want it to. Instead of the index that returns largest order, we would like to find all maximal indices. This also directly altered the behavior of $filterList$ and $filterHelper$ function later. The mistake decreased the success rate in path finding, and lift graphs with respect to more primes could be successfully generated after fixing the function.(ex: $graphLifts(11)$ could be generated after the fix.)

Algorithm 19 findMaxOrd(List, p): find the index with maximum order and return a list of indices with maximum order

Require: $len(List) == 3, p > 0$
$maxOrder = max(findOrder(List, i, p)), i \in \{0, 1, 2\}$
$indexList = [indices\ that\ gives\ maxOrder]$
$return\ indexList$

revised findMaxOrd
note that $checkCage$ and $findMaxOrd$ could also be combined as their functionalities are similar.

Algorithm 20 findMaxOrd(List, p): given a point, return the list of maximal indices with respect to the point

---

Require: $len(List) = 3, p$ *positive index*
$returnList = []$ **for** *i in range(2)* **do**
  if $checki(List, i, p) > 0$ **then**
    $returnList+ = [i]$
  end
end
return returnList

Algorithm 21 checkCage(List, p: given a triple and a prime p, returns the maximal index if the triple is in the cage, returns -1 if the point is not in the cage

---

Require: $len(List) == 3, p > 0$

Ensure: returns maximal index if the point is in the cage, -1 if point is not in the cage
$indList = findMaxOrd(List, p)$ ▷ find a list of all maximal indices
if $len(indList) == 0$ **then**
  $return\ -1$
end
▷ if there is no maximal index, then the point is not in the cage
$maxOrdInd = (maximal\ index\ in\ indList\ that\ has\ maximum\ order)$
$return\ -1$

Algorithm 22 findCage(p): given a prime p, returns the set of triples that are in the cage

---

Require: $p \geq 0$
$allTriple = findTriples(p)$
$cage = []$
for *List in allTriple* **do**
  if $checkCage(List, p) \geq 0$ **then**
    $cage+ = [List]$
  end
end
return cage

note: skipped find sqrt function

note: for now, I skipped functions that highlight different types of points in the cage/graphing percentag of points in cage functions as they consists mostly collecting data and has less to do with algorithm. I included the graphs in the data writeup

BGS algorithm

The ultimate goal of the following functions is to find a path between two points in G_p as outlined in BGS

Algorithm 23 findPathFromOrigin(p): returns a list of 2 elements $[i, k]$ such that $Rot_i^k(1,1,1)$ is in the cage

---

Require: $p > 0$

initial = [1, 1, 1]

for k *in range* $p^2 + 1$ **do**

  initial = Rot1 (initial) ▷ apply Rotation 1 and check if the result is in the cage

  if $checkCage(initial, p) \geq 0$ **then**

    $return[1, k+1]$

  end

end

initial = [1, 1, 1]

for k *in range* $p^2 + 1$ **do**

  initial = Rot2 (initial) ▷ apply Rotation 2 and check if the result is in the cage

  if $checkCage(initial, p) \geq 0$ **then**

    $return[2, k+1]$

  end

end

initial = [1, 1, 1]

for k *in range* $p^2 + 1$ **do**

  initial = Rot3 (initial) ▷ apply Rotation 3 and check if the result is in the cage

  if $checkCage(initial, p) \geq 0$ **then**

    $return[3, k+1]$

  end

end

Note that currently, the function doesn't output the optimal(shortest) path. A few lines of code could be added and output the optimal path if one wants to.

Also, the form of path($[i, k]$ form) in this function is inconsistent with what is used in some later functions(I used strings to represent paths later). This is mainly because I thought this form would be more convenient to convert and apply to points. It turns out that using strings to represent path is more readable.

The following set of functions are mainly used to find a path between two points that are both in the cage

some notes so that I don't forget the overall framework.

The following function finds the intermediate points that connects given cage points $Point1, Point2$. Given $(x_1, x_2, x_3), (y_1, y_2, y_3)$ such that $ord_p(x) = ord_{p,i}(x), ord_p(y) = ord_{p,j}(y), i = 1, j = 2$, we try to find $X' = (x_1, \gamma, z), Y' = (\zeta, y_2, z) \in X^*(p)$. This allows us to connect X to X',

Algorithm 24 findInverse(n, p): find the multiplicative inverse of $n \mod p$

Require: $n > 0, p$ *positive prime* **for** *i in range (p)* **do**

 if $n * i \mod p = 1$ **then**

  return i

 end

end

return -1 ▷ returns -1 if no inverse found

Algorithm 25 checkSolution(List, x, y, p): $List = [\alpha, \beta, z]$, the function checks if α, β, z are solutions to system of equations $(9x^2-4)*(z^2)-\alpha^2 = 4x^2 \mod p$ and $(9y^2-4)*(z^2)-\beta^2 = 4y^2 \mod p$

Require: $len(List) = 3, x > 0, y > 0, p$ *positive prime*

 Let $List = [\alpha, \beta, z]$

 if $(9x^2 - 4) * (z^2) - \alpha^2 = 4x^2 \mod p$ *and* $(9y^2 - 4) * (z^2) - \beta^2 = 4y^2 \mod p$ **then**

  return 1

 end

 return 0

Algorithm 26 findSol(x, y, p:Given constants x, y find all solutions to system of equations $(9x^2 - 4) * (z^2) - \alpha^2 = 4x^2 \mod p, (9y^2 - 4) * (z^2) - \beta^2 = 4y^2 \mod p$

Require: $x > 0, y > 0, p\prime$

 $result = []$

 for *all triples [a, b, c] s.t* $1 \leq a \leq p, 1 \leq b \leq p, 1 \leq c \leq p$ **do**

  if *checkSolution[[a, b, c], x, y, p]* **then**

   result+= [a, b, c]

  end

 end

 return result

X' to Y', Y' to Y and find a path between x and y. In the following function, points X', Y' are called middle points. Also note that the following function does not put the values in the right order.

The above function returns tuples of values and corresponding indices. The function below rearranges the tuple $[X', Y', i, j]$ into $[R1, R2]$ s.t the values in the tuple have correct order. To prevent any reassignment/miss-assignment, $R1, R2$ are initialized as $[-1, -1, -1]$. This means that when a mistake in the rearrange function occurs, the require/ensure protocol will fire an alarm.

The following function integrates the previous functions and return the rearranged list of middle points given two initial cage point($Point1, Point2$) that we would like to connect.

Algorithm 27 findMiddlePoints(Point1, Point2, p): given two points in the cage, this function returns a nested list of all possible values of middle points

Require: Point1, Point2 are lists of length 3, p positive prime
Ensure: returns $[[x, \zeta, z], [\gamma, y, z], i, j]$ where $ord_p(Point1) = ord_{p,i}(Point1), ord_p(Point2) = ord_{p,j}(Point2), \zeta, \gamma$ are as described above.

$i = checkCage(Point1, p)$
$j = checkCage(Point2, p)$ ▷ finds maximal index
$x = Point1[i]$
$y = Point2[j]$ ▷ finds values at maximal index of $Point1, Point2$
$returnList = []$
for $solution\ in\ findSol(x, y, p)$ **do**
  $\alpha = solList[0]$
  $\beta = solList[1]$
  $z = solList[2]$
  $inv = findInverse(2, p)$
  $\zeta = (3 * x * z + \alpha) * inv$
  $\gamma = (3 * y * z + \beta) * inv$
  $\zeta = \zeta \mod p$
  $\gamma = \gamma \mod p$
  $xPrime = [x, zeta, z]$
  $yPrime = [\gamma, y, z]$
  $returnList += [(xPrime, yPrime, i, j)]$
end
$return\ returnList$

Algorithm 28 rearrange(xPrime, yPrime, i, j): rearrange the order values in points $xPrime$ and $yPrime$according to indices i and j

Require: $len(xPrime) = len(yPrime) = 3$, $0 \leq i, j \leq 2$, i, j are maximal indices
$let(x, \zeta, z) = xPrime, (\gamma, y, z)$ ▷ assign values to a more readable format
$zindex = [0, 1, 2] \setminus [i, j]$ ▷ find index of z, when$i = j$, pick the smaller index
$R1 = [], R2 = []$ ▷ initialize return sets, $R1, R2$ are sets of length 3
$R1[i] = x, R2[j] = y$ ▷ keep the value at maximal index of both return points
$R1[zindex] = z, R2[zindex] = z$ ▷ initialize the z index
$zetaIndex = [0, 1, 2] \setminus [i, zindex]$
$R1[zetaIndex] = zeta$ ▷ put ζ into the right index of $R1$
$gammaIndex = [0, 1, 2] \setminus [j, zindex]$
$R2[gammaIndex] = gamma$ ▷ put γ into the right index of $R2$
$return\ [R1, R2]$

note: found a mistake of findMaxOrd function when writing the implementation pseudocode – detail above.

Algorithm 29 cagePath(Point1, Point2, p): given two points in the cage, return the rearranged points

Require: $Point1, Point2 \in C(p)$, p positive prime
  $rearrangedList = []$
  for $element\ in\ findMiddlePoints(Point1, Point2, p)$ **do**
    $element = (xPrime, yPrime, i, j)$
    $rearrangedList+ = [rearrange(xPrime, yPrime, i, j), i, j]$
  end
  $return\ rearrangedList$

The following functions ($filterHelper, filterList$) are written to make sure that middle-points generated are in the orbit of each other. Let $X = (x_1, x_2, x_3), Y = (y_1, y_2, y_3)$ be the middle points generated and $x_1 = y_1$, the following functions checks if $x1 = y1$ is maximal at index 1. The function also makes sure that $ord_p(X) \neq ord_{p,1}(X)$, $ord_p(Y) \neq ord_{p,1}(Y)$, (the index that X and Y have in common are maximal, but not the largest maximal index)

Algorithm 30 filterHelper(p1, p2, i, j, p):

Require: $len(p1) = len(p2) = 3; i, j \in \{0, 1, 2\}, p\ positive\ prime$
  $let\ p1 = [x_1, x_2, x_3], p2 = [y_1, y_2, y_3]$
  $ord1 = findMaxOrd(p1, p); ord2 = findMaxOrd(p2, p)$ ▷ find the maximal indices of p1
  for $k\ in\ (ord1 \cap ord2)$ **do**
    if $k \neq i, k \neq j, x_k = y_k$ **then**
      $return\ 1$
    end
  end
  ▷
  $return\ 0$

Algorithm 31 filterList(mPointList, p): given a middle point list in the format of $[[Point1, Point2, i, j], []]$, use $filterHelper$ filter out the middle points that match out requirements

Require: $MList$ is in the form specified above, p positive prime
  iterate through mPointList and filter out the elements that returns 0 when passed into $filterHelper$

We now need functions that outputs a path that connect two points in the orbit of each other.

Algorithm 32 findOrbit(List, i, p): find the i^th orbit of a Markoff triple

Require: $len(List) = 3, i \in \{0, 1, 2\}, p$ *positive prime*
$orbitList = []$
$initialPoint = List$
for k *in range* $(2p)$ **do**
  if $initialPoint \in orbitList$ **then**
    return orbitList
  end
  $orbitList+ = [intialPoint]$
  $initialPoint = Rot_i(initialPoint)$
end
raise exception ▷ if there are more than $2p$ different elements in the orbit, there's a problem

Algorithm 33 findFormula(Point1, Point2, p): given two points that are in the orbit of each other, return the path in the form of string from Point1 to Point2($i.e$: $Point2 = (Path)Point1$)

Require: $len(Point1, POint2) = 3, p$ *positive prime*
$PotentialIndex = \{i \mid Point1[i] = Point2[i]\}$
for i *in PotentialIndex* **do**
  $Orbit = findorbit(Point1, i, p)$
  check if Point 2 is in the orbit of Point1, if it is, return Rot_i^{k+1} where $k =$ *index of Point*2 *in Orbit*
end
return -1

In $findFormula$, if no formula is found, -1 is returned instead of raising exception. This is because we don't want to terminate the running process due to failure of $findFormula$. Instead, in the $findPath$ function, we will iterate through all possible middlepoint collections before raising exception "path not found".

The following algorithms($findPathHelper, findPath$) are used to find a path between two points in the cage.

Algorithm 34 findPathHelper(nPoint1, nPoint2, MPoint1, MPoint2, p): given two points that we want to connect and two middle points, output a path if there is one

Require: $nPoint1, nPoint2, MPoint1, MPoint2 \in C(p)$
Ensure: $nPoint2 = Path(nPoint1)$
$Path = findFormula(MPoint2, nPoint2, p) + findFormula(MPoint1, MPoint2, p) + findFormula(nPoint1, MPoint1, p)$
if *-1 is in Path* **then**
  return -1
end
▷ checks if $findFormula$ returns a valid a path in every fragment of $Path$
return Path

Algorithm 35 findPath(nPoint1, nPoint2,p): given two points in cage, output a path that connects two points($nPoint2 = Path(nPoint1)$)

Require: $nPoint1, nPoint2 \in C(p), p\ positive\ prime$
Ensure: $nPoint2 = Path(nPoint1)$

```
if findFormula(nPoint1, nPoint2, p)! = −1 then
    return findFormula(nPoint1, nPoint2, p)
end
                        ▷ check if nPoint1 and nPoint2 are in each other's orbit
MPList = filterList(cagePath(nPoint1, nPoint2, p), p)
for mpoint in MPList do
    if findPathHelper(nPoint1, nPoint2, MPoint1, MPoint2, p) ≠ −1 then
        return findPathHelper(nPoint1, nPoint2, MPoint1, MPoint2, p)
    end
end
raise exception 'nopathfound'
```

The next few functions implements middlegame and endgame

Algorithm 36 middleGame(Point, p): given a point with order less than $\sqrt{p}$, output a path that connects the point to a point with order higher than $\sqrt{p}$

Require: $Point \in X^*(p)$
Ensure: $ord((Path)(Point)) > \sqrt{p}$

```
count = 1
while count ≤ p^2 + 1 do
    i = a maximal index
    orbit = findOrbit(Point, i, p)
    maxOrder = max(ord_p(x) | x ∈ orbit)  ▷ find the maximum order all points in the i^th orbit of Point
    maxPoint = y s.t y ∈ orbit, ord_p(y) = maxOrder
    if maxOrder > √p then
        return R_i^k      ▷ if some point in orbit has large enough order, return path that leads to the point
    end
    return middleGame(maxPoint, p) + R_i^k    ▷ R_i^k is the path that connects Point to maxPoint
end
```

Algorithm 37 findRoutetoCage given a point with order > *sqrtp*, return a path that connects the point to cage

this is a brute force algorithm that go through all 3 orbits, find the shorter path from the point to cage respectively, and return the shortest path

The following function combines *middlegame* and $findRoutetoCage$. (There is a bit of abuse of function names here.) The $findPathtoCage$ function checks

Algorithm 38 findPathtoCage(Point, p): given a point not in cage, output a path in form of string that connects it to a point in cage.

Require: $Point \in X^*(p), p\ positive\ prime$
Ensure: $Path(Point) \in C(p)$
if $ord_p(Point) > \sqrt{p}$ **then**
    $return\ findRoutetoCage(Point, p)$
end
$midPath = middleGame(Point, p)$
$intermediateP = applyPath(Point, midPath)$
$endPath = findRutetoCage(intermediateP, p)$
$return\ endPath +'\ ' + midPath$

Algorithm 39 findPathCrossCage(Point1, Point2, p): given two points, one in cage and one not in cage, return a path from $Point1$ to $Point2$ in G_p

combines algorithms 38 and 39 together

The following function puts everything together

Algorithm 40 MarkoffPathFind(Point1, Point2, p): pack all helper functions together and find a path from $Point1$ to $Point2$ on G_p

The whole algorithm could be divided into 1. finding the cage, 2 connect points in the cage, and 3. finding path across the cage.

Note: haven't written the implementation of finding path between parabolic points yet (parabolicPathFind) in code doc . Also, I omitted many functions that performs operations on strings in the form of rotations like $reversePath$ and $applypath$.