procedure sortPipes(pipes[0 to n]: Pipe)

for i := 1 to n

minIndex := i

for j := 1 to n

if pipe[j].getDistance() < pipe[minIndex].getDistance() then minIndex := j

temp := pipes[minIndex]

pipes[minIndex] := pipes[i]

pipes[i] := temp

OU

swap pipes[minIndex] and pipes[i]

procedure find(parent[0 to n]: Integer, i:Integer)

while parent[i] ≠ i

i := parent[i)

return i

procedure union(parent[0 to n]:Integer, rank[0 to n]:Integer, x[0 to n]:Integer, y[0 to n]: Integer)

xRoot := find(parent, x)

yRoot := find(parent, y)

if rank[xRoot] < rank[yRoot] then parent[xRoot] := yRoot

else if rank[xRoot] > rank[yRoot] then parent[yRoot] := xRoot

else parent[yRoot] := xRoot then increment rank[xRoot]

procedure isNotDuplicate(vertices[0 to n ]: text, designation; text)

size := Call methodToReplaceSize with vertices

for i = 0 to size

if vertex = designation then return false

return true

procedure findNumVertices(pipes[0 to n]: Pipe)

vertices = []

size = Call methodToReplaceLength with pipes

for i := 0 to size

designationX := Call get Designation of Call getWaterPoint\_X of pipe

designationY := Call get Designation of Call getWaterPoint\_Y of pipe

if Call isDuplicate with vertices and designationX then

Add designationX to vertices

If Call isDuplicate with vertices and designationY then

Add designationY to vertices

return vertices

procedure kruskalMinSpanningTree(pipes[0 to n]:Pipe)

Call sortPipes with pipes

vertices[0 to n] = Call findNumVertices with pipes

verticeMax := n

minSpanningTree[0 to n]

parent[] := Integer[verticeMax]

rank[] := Integer[verticeMax]

for i := 0 to verticeMax

parent[i] := i

rank[i]:= 0

edgesAdded := 0

pipeIndex := 0

while edgesAdded < verticeMax – 1 and pipeIndex < n

currentPipe := pipes[pipeIndex]

x:= Call methodToReplaceIndexOf with vertices, Call getDesignation of Call getWaterPoint\_X of currentPipe

y:= Call methodToReplaceIndexOf with vertices, Call getDesignation of Call getWaterPoint\_Y of currentPipe

xRoot := Call find with parent and x

yRoot = Call find with parent and y

if xRoot ≠ yRoot then

add currentPipe into minSpanningTree

edgesAdded := edgesAdded + 1

Call union with parent, rank, xRoot, yRoot

pipeIndex := pipeIndex + 1

return minSpanningTree

Procedure findShortestPath(source: SignalPoint, target:SignalPoint, signalPoints[0 to n]:SignalPoint, routes[0 to n]: Route)

numPoints := n (size of signalPoints)

distances[] := Integer[numPoints]

visited[] := Boolean[numPoints]

previous[] := Integer[numPoints]

for i := 0 to numPoints

distances[i] := Maximum value of an Integer

visited[i] := false

previous[i] := -1

sourceIndex := Call methodToReplaceIndexOf with source

distances[sourceIndex] := 0

for i := 0 to numPoint – 1

closest := -1

closestDistance = Maximum value of an Integer

for j := 0 to numPoints

if visited[j] := false and distances [j] < closestDistance then

closest := j

closestDistance := distance[j]

if closest = -1 then

leaves for

visited[closest] := true

for each route in routes

fromIndex := Call methodToReplaceIndexOf with Call getS1 of route

toIndex := Call methodToReplaceIndexOf with Call getS1 of route

if fromIndex = closest and visited[toIndex] 0= false then

newDist = distances[closest] + total distance of route

if newDist < distances[toIndex] then

distances[toIndex] = newDist

previous[toIndex] = closest

if toIndex = closest and visited[fromIndex] = false then

newDist = distances[closest] + total distance of route

if newDist < distances[fromIndex] then

distances[fromIndex] = newDist

previous[fromIndex] = closest

path[0 to n]

for at = Call methodToReplaceIndexOf with target to (at ≠ -1)

add vertex to the begining of the path

if path = null or Call get with 0 of path ≠ source then

return new array

return Call constructRoute with path and routes

procedure constructRoute (signalPoints[0 to n]:SignalPoint, routes[0 to n]:Route)

newRoute[0 to n]

sizeSignalPoint = Call methodToReplaceSize with signalPoints

sizeRoute = methodToReplaceSizeRoute with routes

for i := 0 to sizeSignalPoint - 1

for j = 0 to sizeRoute

if Call get with j of routes == new Route with Call get with i of signalPoints, Call get with i + 1 of signalPoints then

Call add with new Route Call getDistance of Call get with j of routes, Call get with i of signalPoints, Call get with i+1 of signalPoints of newRoute

procedure findShortestPathToNearestAP(source:SignalPoint, signalPoints[0 to n]:SignalPoint, routes[0 to n]:Route, isAssemblyPoint:Boolean)

numPoints = Call methodToReplaceSize with signalPoints

distances[] := Integer[numPoints]

visited[] := Boolean[numPoints]

previous[] = Integer[numPoints]

for i:= 0 to numPoints

distances[i] := Maximum value of Integer

visited[i] := false

previous[i] := -1

sourceIndex := Call methodToReplaceIndexOf of signalPoints with source distances[sourceIndex] := 0

for i := 0 to numPoints -1

closest := -1

closestDistance := Maximum value of an Integer

for j:= 0 to numPoints

if visited[j] = false and distances[j] < closestDistance then

closest := j

closestDistance = distances[j]

if closest = -1 then

leaves for

visited[closest] := true

for each route in routes

fromIndex := Call methodToReplaceIndexOf of signalPoints with route.getS1()

toIndex := Call methodToReplaceIndexOf of signalPoints with route.getS2()

if fromIndex = closest and visited[toIndex] := false then

newDist := distances[closest] + total distance of route

if newDist < distances[toIndex] then

distances[toIndex] := newDist

previous[toIndex] := closest

path[0 to n]

nearestAPIndex := Call findNearestAssemblyPointIndex with signalPoints, distances, isAssemblyPoint

for at :=nearestAPIndex to at ≠ -1

add vertex to the begining of the path

if path = null or first vertex of path ≠ source then

return new array

return Call constructRoute with path and routes

Procedure methodToReplaceIndexOf(vertices, signalPoint):

size = Call methodToReplaceSize with vertices

for i = 0 to size -1

if Call getName of Call get with i of vertices == Call getName of signalPoint

return i

return -1

procedure methodToReplaceSize(array[0 to n]:String):

count = 0

while true

if Call get with count of vértices ≠ null:

count = count + 1

return count

return -1

procedure methodToReplaceSizeRoute(route[0 to n]:Route):

count = 0

while true

if Call get with count of vertices ≠ null:

count = count + 1

return count

return -1

procedure methodToReplaceLength(array[0 to n]:Pipe):

count = 0

while true

if array[count] ≠ null:

count = count + 1

return count

return -1

* **methodToReplaceSizeRoute(route:,array)**
  + Conta os elementos da lista até que ocorra uma IndexOutOfBoundsException.
  + Conta os elementos da lista de rotas.
  + **Complexidade total**: O(n)
* **methodToReplaceLenght(array[0 to n]:, Pipe)**
  + Conta os elementos da lista até que ocorra uma IndexOutOfBoundsException.
  + Conta os elementos da lista de pipes.
  + **Complexidade total**: O(n)
* **methodToReplaceIndexOf(vertices, signalPoint)**
  + Itera sobre a lista vertices até encontrar um elemento que corresponda ao signalPoint ou até o final da lista.
  + Encontra o índice de um ponto de sinalização nos vértices.
  + **Complexidade total**: O(n) + O(n) = O(n).
* **methodToReplaceSize(vértices: , array)**:
  + Conta os elementos da lista até que ocorra uma IndexOutOfBoundsException.
  + Conta os elementos da lista.
  + **Complexidade total**: O(n).
* **sortPipes(pipes[0 to n]: Pipe)**:
  + Implementa uma variação do algoritmo de seleção (selection sort).
  + O loop externo executa n vezes.
  + **Complexidade total**: O(n²).
* **find(parent[0 to n]: Integer, i: Integer)**:
  + Realiza uma caminhada ao longo do array parent até encontrar a raiz.
  + Com a compressão de caminho (path compression), a complexidade amortizada é O(α(n)), onde α é a função inversa de Ackermann.
  + **Complexidade total**: O(α(n)).
* **union(parent[0 to n]: Integer, rank[0 to n]: Integer, x[0 to n]: Integer, y[0 to n]: Integer)**:
  + Encontra as raízes de x e y usando a função find, que é O(α(n)) em média.
  + Executa comparações e possivelmente atualiza o array parent e rank, que são operações O(1).
  + **Complexidade total**: O(α(n)).
* **isNotDuplicate(vertices[0 to n ]: text, designation: text):**
  + Verifica se a designação já existe no array de vértices.
  + **Complexidade total**: O(n).
* **findNumVertices(pipes[0 to n]: Pipe)**:
  + Itera sobre todos os pipes para encontrar os vértices únicos.
  + Para cada pipe, chama isDuplicate, que é O(n) no pior caso.
  + **Complexidade total**: O(n²).
* **constructRoute(signalPoints[0 to n]: SignalPoint, routes[0 to n]: Route)**:
  + Itera sobre signalPoints e verifica cada route, o que pode resultar em O(n²).
  + Constrói uma nova rota a partir de pontos de sinalização e rotas fornecidas.
  + **Complexidade total**: O(n²).
* **findShortestPath(source: SignalPoint, target: SignalPoint, signalPoints[0 to n]: SignalPoint, routes[0 to n]: Route):**
  + Inicializa arrays e busca o índice da fonte em O(n).
  + O loop principal executa n vezes.
  + Implementa o algoritmo de Dijkstra para encontrar o caminho mais curto entre dois pontos.
  + Cada iteração do loop interno também pode executar n vezes, resultando em O(n²) no total.
  + **Complexidade total**: O(n²).
* **findShortestPathToNearestAP(source: SignalPoint, signalPoints[0 to n]: SignalPoint, routes[0 to n]: Route, isAssemblyPoint: Boolean)**:
  + Muito similar a findShortestPath, com uma chamada adicional a findNearestAssemblyPointIndex, que também é O(n). A complexidade é dominada pelo algoritmo de Dijkstra.
  + **Complexidade total**: O(n²).
* **kruskalMinSpanningTree(pipes[0 to n]: Pipe):**
  + Chama sortPipes, que é O(n²).
  + Chama findNumVertices, que é O(n²).
  + Inicializa arrays parent e rank em O(n).
  + Itera sobre os pipes e faz operações de união e busca, que são O(nα(n)) no total, onde α é a função inversa de Ackermann.
  + **Complexidade total**: O(n²) (dominado pela ordenação e pela busca de vértices únicos).