Benjamin Duncan - Lab 3 Network Routing

CS 312 - Winter 2023

Part 1 - Dijkstra's Algorithm Code Implementation

See Appendix 1 -> NetworkRoutingSolver.py -> dijkstras()

Part 2 - Priority Queue Implementations

See Appendix 1 -> PriorityQueue.py

Part 3 - Time and Space Complexity

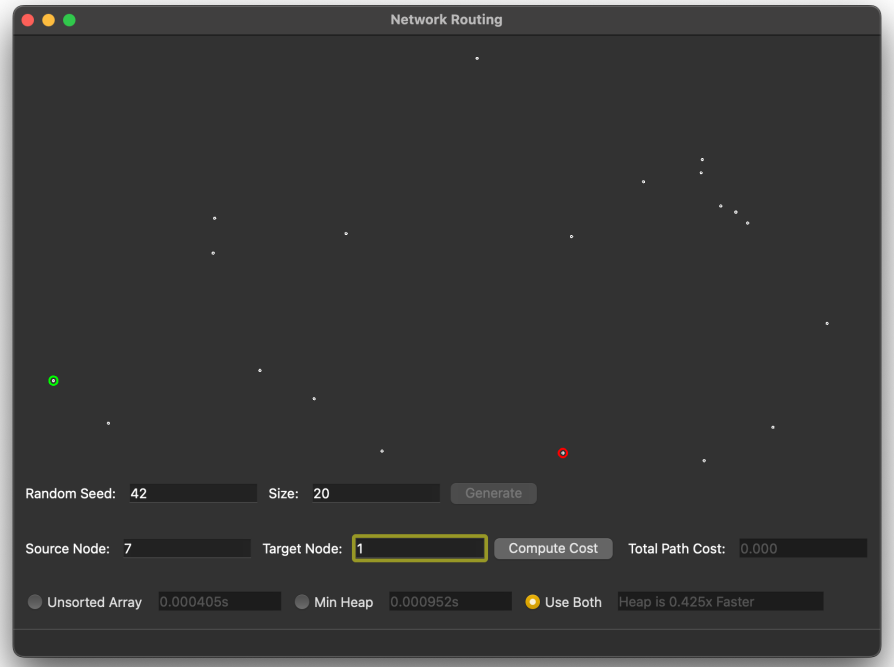
There are 7 significant functions that need to be analyzed for their time and space complexity. Each function in each of the priority queue implementations has the time and space complexity written in comments above it. Similarly, the *dijkstras* function has its time and space complexity written above it too. For each of the priority queue implementations, the *makeQueue, decreaseKey,* and *deleteMin* functions are of import.

- dijkstras()
    - Time complexity: $O(|V| * max(Dm, Dk) + |E|)$ - The time complexity of dijkstras algorithm depends greatly upon the priority queue implementation. Because we iterate through each vector once and each edge once, calling *decreaseKey* and *deleteMin* once for each vector, we multiply |V| by max(Dm, Dk), where Dm = *deleteMin* and Dk = *decreaseKey*. Because we also need to iterate through each edge and compare weights, we add |E| to the time complexity.
    - Space complexity: O(max(Dm, Dk)) - Depending on the implementation of the priority queue, the space complexity can vary greatly. The array implementation doesn't have any recursive functions in it, resulting in most of the functions being O(1) space complexity. The only function that I marked as having O(n) space complexity in the array implementation was the makeQueue function because it needs to assign the initial arrays. In the binary heap implementation of the priority queue, *decreaseKey* and *deleteMin* both have recursive executions, resulting in O(log n) space complexity. So, because the space complexity depends upon the priority queue implementation and the other code within the *dijkstras* function is O(1) space complexity, the total space complexity is O(max(Dm, Dk)).
- Priority queue - **heap** implementation
    - makeQueue
        - Time complexity: O(n) - the time complexity of this function is O(n) because we need to iterate through the arrays and assign the appropriate values.

        - Space complexity: O(n) - the space complexity of this function is O(n) because of the need to initiate the arrays that scale to the size of |V|.
    - decreaseKey
        - Time complexity: O(log n) - The time complexity of *decreaseKey* is O(log n) because of the need to possibly bubble up a node in the binary heap once its value has been changed. A recursive function is used to bubble up the node.
        - Space complexity: O(log n) - The space complexity of *decreaseKey* is O(log n) because the function to bubble up the node once its value has changed is recursive, going up the binary heap tree which has a max depth of log(|V|) base 2.
    - deleteMin
        - Time complexity: O(log n) - The time complexity of deleteMin is affected in large part by the recursive function that replaces the root node once it has been deleted. Because the tree depth is only log(|V|) base 2, the algorithm is also O(log n) time complexity.
        - Space complexity: O(log n) - The space complexity of deleteMin is affected in large part by the recursive function that replaces the root node once it has been deleted. This is very similar to the time complexity.. Because the tree depth is only log(|V|) base 2, the algorithm is also O(log n) space complexity.
- Priority queue - **unsorted array** implementation
    - makeQueue
        - Time complexity: O(n) - this is affected by the amount of vectors because this function defines the arrays used in this implementation.
        - Space complexity: O(n) - this is affected by the amount of vectors because this function defines the arrays used in this implementation.
    - decreaseKey
        - Time complexity: O(1) - this is a simple assignment
        - Space complexity: O(1) - this is a simple assignment
    - deleteMin
        - Time complexity: O(n) - this is affected by a for-loop which searches out the lowest value and returns it.
        - Space complexity: O(1) - there is a constant use of memory in this algorithm.

Part 4 - Examples

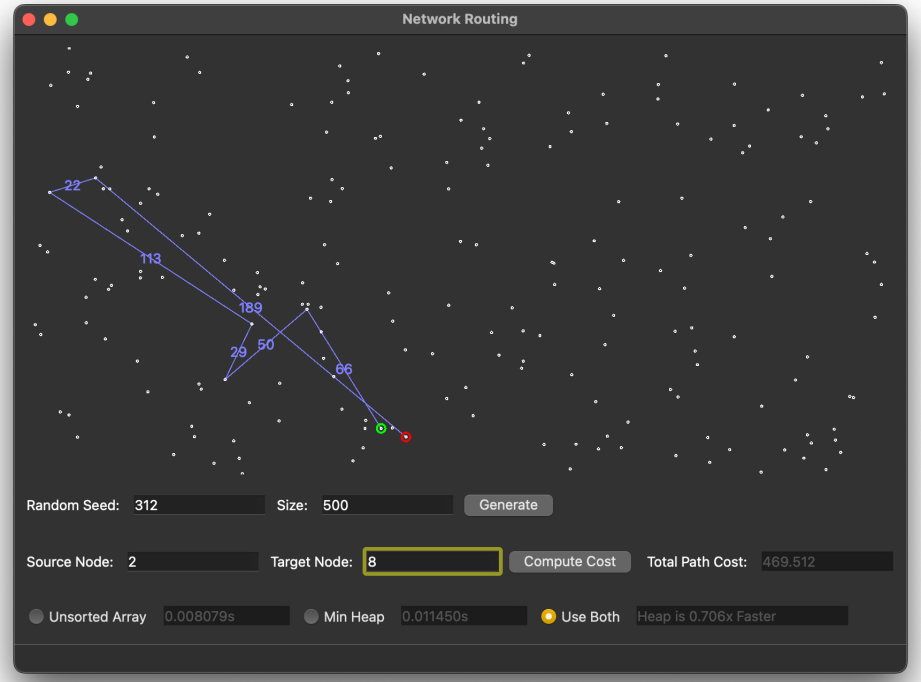
A.


Network Routing
Random Seed: 42
Size: 20
Generate
Source Node: 7
Target Node: 1
Compute Cost
Total Path Cost: 0.000
Unsorted Array
0.000405s
Min Heap
0.000952s
Use Both
Heap is 0.425x Faster

B.

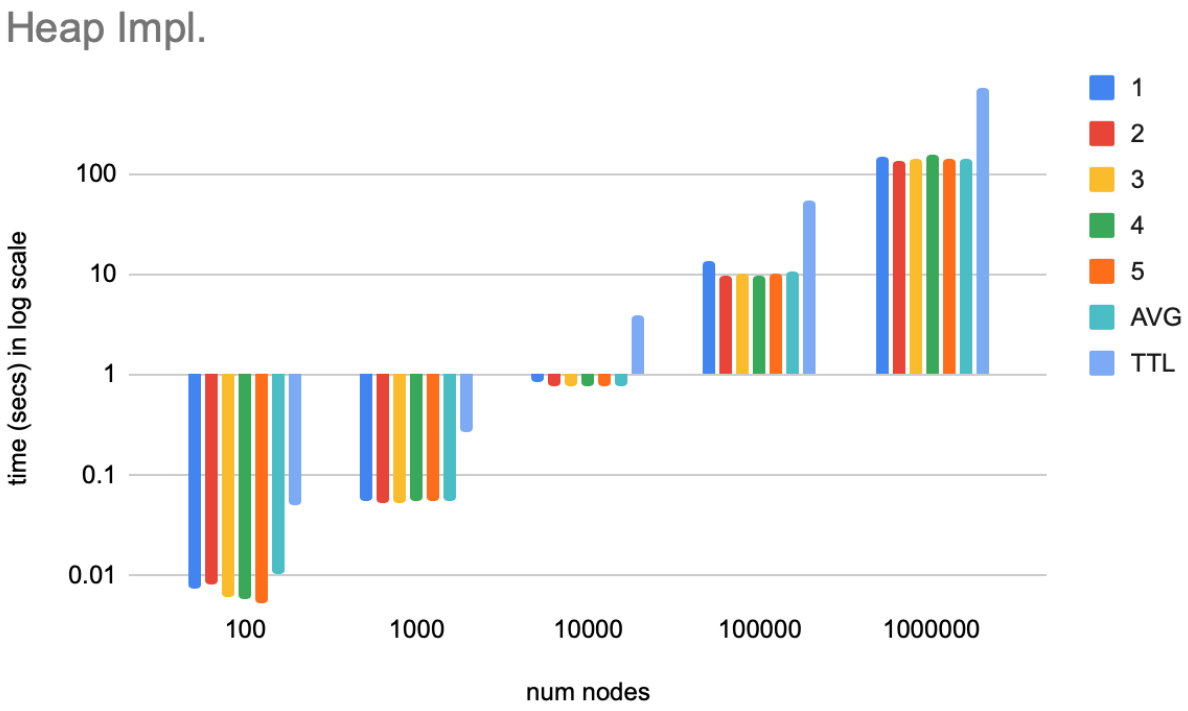

Network Routing
40
193
137
58
242
39
43
158
Random Seed: 123
Size: 200
Generate
Source Node: 94
Target Node: 3
Compute Cost
Total Path Cost: 911.081
Unsorted Array
0.005174s
Min Heap
0.009539s
Use Both
Heap is 0.542x Faster

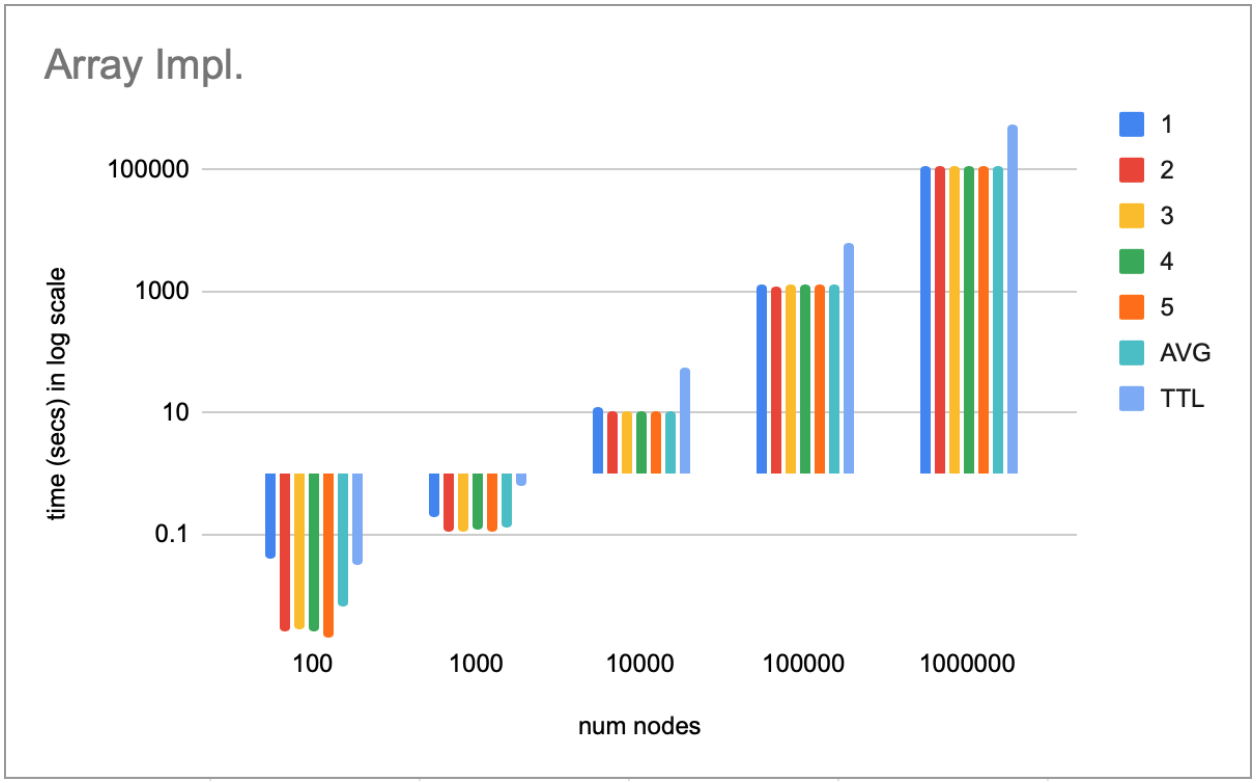
C.



Part 5

The difference between the array and heap implementations is pretty darn great. It's typically on a magnitude of 10, meaning that the hepa implementation was generally 10x faster than the array implementation. For the times of the array implementation where n = 1,000,000, I estimated the time to be about 111,111 by graphing the previous n vs average and creating a function that went through all the points. The intersection at x = 1,000,000 was then the estimated value. It holds up to be about 10x the value of the heap implementation at n = 1,000,000 which was 142 secs. I'm glad that I didn't run the program because it would've taken 30 hours total, according to the estimate. I'm not sure that my computer would've had enough memory to run it! It's very interesting to see the benefits of a binary heap implementation in a real scenario.


Heap Impl.
time (secs) in log scale
100
10
1
0.1
0.01
100
1000
10000
100000
1000000
num nodes
1
2
3
4
5
AVG
TTL


Array Impl.
time (secs) in log scale
100000
1000
10
0.1
100
1000
10000
100000
1000000
num nodes
1
2
3
4
5
AVG
TTL

Appendix

NetworkRoutingSolver.py

```
#!/usr/bin/python3
from PriorityQueue import PriorityQueue, PriorityQueueHeap, PriorityQueueArray
from CS312Graph import *
import time
import array

array.array('i')


class NetworkRoutingSolver:
   prev = None
   dist = None

   def __init__(self):
       pass

   def initializeNetwork(self, network):
       assert (type(network) == CS312Graph)
       self.network = network

   def getShortestPath(self, destIndex):
       self.dest = destIndex
       path_edges = []
       total_length = 0
       nodeId = destIndex
       prevNodeId = self.prev[nodeId]
       while prevNodeId != -1:
           prevNode = self.network.nodes[prevNodeId]

           edge = None
           for e in prevNode.neighbors:
               if e.dest.node_id == nodeId:
                   edge = e
                   break

           if edge is None:
               print("Error: no edge found.")
               break

           path_edges.append((edge.src.loc, edge.dest.loc, '{:.0f}'.format(edge.length)))
           total_length += edge.length

           nodeId = prevNodeId
           prevNodeId = self.prev[nodeId]
       return {'cost': total_length, 'path': path_edges}

   # Time & space complexity same as dijkstras()
   def computeShortestPaths(self, srcIndex, use_heap=False):
       self.source = srcIndex
       t1 = time.time()
```

```
       (self.prev, self.dist) = self.dijkstras(srcIndex, self.network.nodes,
PriorityQueueHeap() if use_heap else PriorityQueueArray())

       t2 = time.time()
       return t2 - t1

   # Time: O(|V| * max(Dm, Dk) + |E|)
   # Space: O(max(Dm, Dk))
   # Dm = decreaseMin
   # Dk = decreaseKey
   def dijkstras(self, srcIndex, nodes, q: PriorityQueue) -> (array.array, array.array):
       numNodes = len(self.network.nodes)
       prev = array.array('i', (-1 for i in range(0, numNodes)))

       q.makeQueue(nodes)
       q.decreaseKey(srcIndex, 0)

       while not q.isEmpty():
           nodeId = q.deleteMin()
           node = nodes[nodeId]
           curDist = q.getDist(nodeId)
           for edge in node.neighbors:
               if q.getDist(edge.dest.node_id) > curDist + edge.length or
q.getDist(edge.dest.node_id) == q.INF:
                   q.decreaseKey(edge.dest.node_id, curDist + edge.length)
                   prev[edge.dest.node_id] = nodeId

       dist = q.getKeys()
       return prev, dist
```

PriorityQueue.py

```
import array
import math

array.array('i')


class PriorityQueue:
   INF = 1000000000
   DNE = -1
   keys: array = None

   def makeQueue(self, nodes):
       pass

   def decreaseKey(self, index, value):
       pass

   def deleteMin(self) -> int:
       pass

   def isEmpty(self) -> bool:
       pass

   def getKeys(self) -> array.array:
       pass

   def getDist(self, nodeId) -> float:
       pass
```

```
class PriorityQueueHeap(PriorityQueue):
   heap: array = None
   point: array = None
   keys: array = None
   finalkeys: array = None

   def __init__(self):
       pass

   # Time: O(n)
   # Space: O(n)
   def makeQueue(self, nodes):
       self.heap = array.array('i', (node.node_id for node in nodes))
       self.point = array.array('i', (node.node_id for node in nodes))
       self.keys = array.array('f', (self.INF for i in range(0, len(nodes))))
       self.finalkeys = array.array('f', (self.INF for i in range(0, len(nodes))))

   # Time: O(log n)
   # Space: O(log n)
   def decreaseKey(self, nodeId: int, value: float):
       self.keys[nodeId] = value
       self.finalkeys[nodeId] = value
       self.bubbleUp(self.getHeapIndex(nodeId))

   # Time: O(log n)
   # Space: O(log n)
   def deleteMin(self) -> int:
       minIndex = 0
       minNodeId = self.heap[minIndex]
       self.finalkeys[self.getNodeId(minIndex)] = self.keys[self.getNodeId(minIndex)]
       self.keys[self.getNodeId(minIndex)] = self.DNE
       self.bubbleDown(minIndex)
       return minNodeId

   # Time: O(log n)
   # Space: O(log n)
   def bubbleUp(self, heapIndex):
       hi = heapIndex
       if hi == 0:
           return
       pi = self.getParentIndex(hi)
       if self.doSwapNodes(pi, hi):
           self.swapNodes(pi, hi)
           self.bubbleUp(pi)

   # Time: O(log n)
   # Space: O(log n)
   def bubbleDown(self, heapIndex):
       pi = heapIndex
       fi = math.floor(self.getChildIndex(pi))  # first child index

       if fi >= len(self.heap):
           pass
       elif fi + 1 >= len(self.heap) and self.doSwapNodes(pi, fi):
           self.swapNodes(pi, fi)
       elif fi + 1 >= len(self.heap):
           pass
       elif self.getNodeWeightByHI(fi) < self.getNodeWeightByHI(fi + 1) and
self.doSwapNodes(pi, fi):
           self.swapNodes(pi, fi)
           self.bubbleDown(fi)
```

```
        elif self.doSwapNodes(pi, fi + 1) and not (self.getNodeWeightByHI(fi) >= 0 >
self.getNodeWeightByHI(fi + 1)):
            self.swapNodes(pi, fi + 1)
            self.bubbleDown(fi + 1)
        elif self.doSwapNodes(pi, fi):
            self.swapNodes(pi, fi)
            self.bubbleDown(fi)

    # Time: O(1)
    # Space: O(1)
    def doSwapNodes(self, parentHeapIndex, childHeapIndex) -> bool:
        if parentHeapIndex >= len(self.keys) or childHeapIndex >= len(self.keys):
            return False

        pw = self.getNodeWeightByHI(parentHeapIndex)
        cw = self.getNodeWeightByHI(childHeapIndex)
        return (pw > cw or (pw < 0 <= cw)) and not (pw >= 0 > cw)

    # Time: O(1)
    # Space: O(1)
    def swapNodes(self, heapI1, heapI2):
        nodeId1 = self.getNodeId(heapI1)
        nodeId2 = self.getNodeId(heapI2)

        (self.heap[heapI1], self.heap[heapI2]) = (self.heap[heapI2], self.heap[heapI1])
        (self.point[nodeId1], self.point[nodeId2]) = (self.point[nodeId2], self.point[nodeId1])

    # Time: O(1)
    # Space: O(1)
    def getNodeWeight(self, nodeId):
        return self.keys[nodeId]

    # Time: O(1)
    # Space: O(1)
    def getNodeWeightByHI(self, heapIndex):
        return self.getNodeWeight(self.getNodeId(heapIndex))

    # Time: O(1)
    # Space: O(1)
    def getHeapIndex(self, nodeId):
        return self.point[nodeId]

    # Time: O(1)
    # Space: O(1)
    def getNodeId(self, heapIndex):
        return self.heap[heapIndex]

    # Time: O(1)
    # Space: O(1)
    def getChildIndex(self, index) -> int:
        return index * 2 + 1

    # Time: O(1)
    # Space: O(1)
    def getParentIndex(self, index) -> int:
        return math.floor((index - 1) / 2)

    # Time: O(1)
    # Space: O(1)
    def isEmpty(self) -> bool:
        return self.getNodeWeightByHI(0) < 0
```

```
    # Time: O(1)
    # Space: O(1)
    def getKeys(self) -> array.array:
        return self.finalkeys

    # Time: O(1)
    # Space: O(1)
    def getDist(self, nodeId) -> float:
        return self.finalkeys[nodeId]

    # I didn't calculate the complexity of this method because I only used it for debugging
    def print(self):
        index = "index: "
        for i in range(0, len(self.keys)):
            index += str(i) + " "
        points = "point:  "

        heap = "heap:   \n\t "
        for k in range(0, math.floor(math.log(len(self.heap), 2) - 1)):
            heap += " "
        heap += str(self.heap[0]) + "\n\t"
        i = 0
        j = 0
        for nodeId in self.heap:
            if j == 0:
                j = 1
                continue
            for k in range(0, math.floor(math.log(len(self.heap), 2) - j)):
                heap += " "
            heap += str(nodeId)
            if i == j * 2 - 1:
                i = 0
                j += 1
                heap += "\n\t"
            else:
                heap += " "
                i += 1

        keys = "keys:   "
        for dist in self.keys:
            keys += (str(dist) if dist >= 0 else ('DNE' if dist == -1 else 'INF')) + " "
        for point in self.point:
            points += str(point) + "   "
        print(index)
        print(keys)
        print(points)
        print(heap)


class PriorityQueueArray(PriorityQueue):
    queue: array = None
    keys: array = None

    def __init__(self):
        pass

    # Time: O(n)
    # Space: O(n)
    def makeQueue(self, nodes):
        self.queue = array.array('i', (node.node_id for node in nodes))
        self.keys = array.array('f', (self.INF for i in range(0, len(nodes))))
```

```
    # Time: O(1)
    # Space: O(1)
    def decreaseKey(self, nodeId: int, value: float):
        self.keys[nodeId] = value

    # Time: O(n)
    # Space: O(1)
    def deleteMin(self) -> int:
        minIndex = 0
        minWeight: int = self.keys[self.queue[0]]
        for i in range(0, len(self.queue)):
            if (self.keys[self.queue[i]] < minWeight and self.keys[self.queue[i]] != self.INF)
or minWeight < 0:
                minIndex = i
                minWeight = self.keys[self.queue[i]]
        return self.queue.pop(minIndex)

    # Time: O(1)
    # Space: O(1)
    def isEmpty(self) -> bool:
        return len(self.queue) == 0

    # Time: O(1)
    # Space: O(1)
    def getKeys(self) -> array.array:
        return self.keys

    # I didn't calculate the complexity of this method because I only used it for debugging
    def print(self):
        print("\n\n")
        index = "index: "
        queue = "queue: "
        keys = "keys:  "
        for i in range(0, len(self.keys)):
            index += str(i) + " "
        for nodeId in self.queue:
            queue += str(nodeId) + " "
        for dist in self.keys:
            keys += str(dist) + " "
        print(index)
        print(queue)
        print(keys)

    # Time: O(1)
    # Space: O(1)
    def getDist(self, nodeId) -> float:
        return self.keys[nodeId]
```