I highly recommend viewing the code on our github instead

**main.py**

```
from agent import Agent


def main():
    # Question 6, running for n=100 orders
    n = 100
    consecutive_agent = Agent()                                    # Initialize
agent
    for i in range(n):                                             # 100 orders
        consecutive_agent.protocol()                               # Run protocol

    # Question 5
    edge_case_agent = Agent()                                      # Initialize
agent
    edge_case_agent.protocol(rand=False, shelves=[33],             # Passing in our
own parameters for edge case
                             div=6, single_run=True)


if __name__ == '__main__':
    main()
```

**agent.py**

```
from project_utils import path_trace, path_merge, path_forger
from warehouse import Warehouse, Division, Order


class Agent:
    def __init__(self):
# Agent initialization
        # Warehouse Components
        self.order = None
        self.location = {
            'warehouse': 1,
            'div': 1
        }
        self.wh = Warehouse()
        self.div = Division()

        # Scoring and Tracking
        self.score = 0
        self.cumulative_score = 0
        self.num_runs = 0
        self.step_count = 0
        self.cumulative_steps = 0
        self.max_path = float('-inf')
        self.min_path = float('inf')

    def protocol(self, rand=True, shelves=None, div=None, single_run=False):
# Main protocol
        self.order = Order(rand, shelves, div)
```

```
# Generate the order for this run
        self.wh_search(self.order.div, single_run)
# Search the warehouse for the division
        self.div_search(self.order.shelves, single_run)
# Enter division and search for order
        if single_run:
            print("Returning to Warehouse")
# Printouts for single run
            print(f'Final Score: \t{self.score}')
            print(f'Final Step Count: {self.score}')
            self.score = 0
            self.step_count = 0
        else:
# Printouts for consecutive runs
            self.scoring_func()

    def wh_search(self, target_node, single_run):
# Finding the division in warehouse:
        target_path = path_merge(self.location['warehouse'], target_node)
# Merge path agent to target
        self.move(target_path, loc='warehouse', single_run=single_run)
# Move on the merged path

    def div_search(self, shelves, single_run):
# Finding shelves in a division:
        agent_path = path_trace(self.location['div'])
# Trace agent back to root

        targets = self.id_search(shelves)
# Iterative Deepening Search -> Nodes
        target_paths = path_forger(targets, agent_path)
# Create priority queue

        while len(target_paths) > 0:
# Until targets have all been retrieved
            this_path = target_paths[0]
# Get shortest path
            self.move(this_path, loc='div', single_run=single_run)
# and move there
            agent_path = path_trace(self.location['div'])
# Get agent path to assess next target
            del targets[f'{this_path[-1]}']
# Delete this target from targets
            target_paths = path_forger(targets, agent_path)
# Recalculate paths to targets
        self.go_home(single_run)
# Return to division root

    def move(self, path, loc, single_run):
# Move along a path
        current_tree = self.wh.node_list if loc == 'warehouse' \
            else self.div.node_list
# Define current tree
        for step in path:
            current_node = current_tree[self.location[loc] - 1]
# Get current node
            if step == self.location[loc]:
```

```
                continue
# Don't step if already there
            self.step_count += 1
# Step counter
            self.location[loc] = step
# Move agent
            if loc == 'div':
                self.score += 1
# Division weights = 1
            else:
                next_node = self.wh.node_list[step - 1]
# Get next node
                self.score_calc(current_node, next_node)
# Score weight for moving along edge
            if single_run:
# Printouts for individual runs
                print(f'Moved to {self.location[loc]} in {loc}, current
score: {self.score}')

    def score_calc(self, node1, node2):
        parent_node = node1 if node2.parent == node1.num else node2
# Figure out which node is parent
        child_node = node1 if parent_node != node1 else node2
# Other node is child

        edge = parent_node.left if parent_node.left[0] == child_node.num \
            else parent_node.right
# Retrieve edge weight

        self.score += edge[1]
# Add edge weight to score

    def id_search(self, targets):
# Iterative Deepening Search
        results = {}
        for target in targets:
# Loop through current targets
            results[f'{target}'] = 'None'
# No initial result
            depth = 0
            while depth <= self.div.max_depth:
# Incrementing depth limit for DLS
                result = self.dl_search(target, depth)
# Try DLS at current depth limit
                if result:
                    results[f'{target}'] = result
# If target is found, add to results
                    break
# and break out of DLS loop
                depth += 1
# Otherwise, increment depth
        return results

    def dl_search(self, target, depth_limit):
        frontier = list()
# Empty frontier
        frontier.append(self.div.node_list[0])
```

```
# Start at root
        while len(frontier) > 0:
# Until frontier is exhausted
            node = frontier.pop()
# Pop top node
            if node.num == target:
                return node
# Return node if target found
            if not node.depth > depth_limit:
# If within depth limit
                self.expand(frontier, node)
# expand frontier from node
        return None
# If you make it this far, search failed

    def expand(self, frontier, node):
        right_index = node.right[0] - 1
# Get frontier node indices
        left_index = node.left[0] - 1

        frontier.append(self.div.node_list[right_index])
# Add nodes to frontier
        frontier.append(self.div.node_list[left_index])

    def go_home(self, single_run):
        self.move(
# Move back to division root
            path_trace(self.location['div'], reverse=False),
            loc='div', single_run=single_run
        )

    def scoring_func(self):
        # Score calculations
        self.num_runs += 1
        self.cumulative_score += self.score
        average_score = (self.cumulative_score + self.score) / self.num_runs

        # Step calculations
        self.cumulative_steps += self.step_count
        average_steps = (self.cumulative_steps + self.step_count) /
self.num_runs
        self.max_path = self.step_count if self.max_path < self.step_count
else self.max_path
        self.min_path = self.step_count if self.min_path > self.step_count
else self.min_path

        # Report printout
        print(f'Run number {self.num_runs}:')
        print('########################################')
        print(f'Score this run: \t{self.score}')
        print(f'Average Score: \t\t{round(average_score, 2)}\n')

        print(f'Avg Step Count: \t{round(average_steps)}')
        print(f'Max Step Count: \t{self.max_path}')
        print(f'Min Step Count: \t{self.min_path}')
        print('########################################')
```

```
        # Reset non-cumulative metrics
        self.score = 0
        self.step_count = 0
```

**warehouse.py**

```
import math
import random

from project_utils import path_trace


# WAREHOUSE MAIN FLOOR
class Node:
    def __init__(self, num, is_parent, left=None, right=None):
        # Warehouse info
        self.num = num                                                  #
Reference number
        self.depth = math.floor(math.log2(num))                         #
Depth
        self.root_path = path_trace(num)                                #
Path from node to root

        # Children edges (if any) in tuple form (node, weight)
        self.is_parent = is_parent                                      # Has
children
        self.left = left
        self.right = right

        # Parent node (except for root)
        self.parent = num // 2 if num > 1 else None                     #
Parent node


def floor_gen():
    node_list = list()

    # Node list hard-coded in due to nature of tree.
    node_list.append(Node(1, is_parent=True, left=(2, 20), right=(3, 20)))
    node_list.append(Node(2, is_parent=True, left=(4, 20), right=(5, 30)))
    node_list.append(Node(3, is_parent=True, left=(6, 40), right=(7, 10)))
    node_list.append(Node(4, is_parent=True, left=(8, 10), right=(9, 20)))
    node_list.append(Node(5, is_parent=True, left=(10, 30), right=(11, 20)))
    node_list.append(Node(6, is_parent=True, left=(12, 30), right=(13, 20)))
    node_list.append(Node(7, is_parent=True, left=(14, 20), right=(15, 20)))
    node_list.append(Node(8, is_parent=False))
    node_list.append(Node(9, is_parent=False))
    node_list.append(Node(10, is_parent=False))
    node_list.append(Node(11, is_parent=False))
    node_list.append(Node(12, is_parent=False))
    node_list.append(Node(13, is_parent=False))
    node_list.append(Node(14, is_parent=False))
    node_list.append(Node(15, is_parent=False))

    return node_list
```

```
class Warehouse:
    def __init__(self):
        self.node_list = floor_gen()                                    #
Generate Warehouse tree


# WAREHOUSE DIVISION
def generate_tree(max_node=63):
    node_list = list()                                                  #
Division generator
    i = 1
    while i <= max_node:
        if i <= max_node // 2:                                          #
Nodes with children
            left_child = (i * 2, 1)                                     #
Generate left child
            right_child = (i * 2 + 1, 1)                                #
Generate right child
            node_list.append(Node(i, True, left_child, right_child))    #
Create Node based on info and add to list
        else:
            node_list.append(Node(i, False))                            #
Create a leaf Node and add to list
        i += 1
    max_depth = math.floor(math.log2(max_node))                         #
Leaf depth
    return node_list, max_depth


class Division:
    def __init__(self):
        self.node_list, self.max_depth = generate_tree()                #
Generate Division tree


# ORDER GENERATION
class Order:
    def __init__(self, rand=True, shelves=None, div=None):
        if rand:                                                        #
Random generator if rand
            self.shelves = random.sample(range(1, 63), random.randint(1, 3))
            self.div = random.randint(1, 15)
        else:                                                           #
Pass in parameters if not rand
            self.shelves = shelves
            self.div = div
```

**project_utils.py**

```
import math


# PATH UTILITIES
def path_trace(node, reverse=True):
# Root traceback
```

```
    level = math.floor(math.log2(node))
# Current depth
    root_path = [node]
# Start from this node
    next_node = node
    for i in range(level):
        next_node = next_node // 2
# Step back to parent,
        root_path.append(next_node)
# add it to the path list
    if reverse:
        root_path.reverse()
# reverse the order so root is first
    return root_path


def path_merge(var1, var2, method='node'):
# Path merging utility
    path1, path2, common = find_common(var1, var2, method=method)
# Find common node
    path1.reverse()
# Because the agent is backtracking

    path1_merge_point = path1.index(common)
# Find index of common node
    path2_merge_point = path2.index(common)
# Find index of common node

    new_path = path1[0:path1_merge_point] + path2[path2_merge_point:]
# Merge the two at the common node

    return new_path
# Return merged path


def find_common(var1, var2, method='node'):
# Utility to find common node
    if method == 'node':
# If passed two nodes,
        path1 = path_trace(var1)
# run path trace on each
        path2 = path_trace(var2)
# to get their root paths.
    else:
        path1 = var1
# Otherwise, use the paths passed in
        path2 = var2
    rng = min(len(path1), len(path2))
# Use shorter path length

    common_node = 1
# Root
    for i in range(rng):
        if path1[i] == path2[i]:
# Change common if they are the same
            common_node = path1[i]
        else:
```

```
# Otherwise, the paths have diverged
            break

    return path1, path2, common_node
# Return the paths and their common node


def path_forger(targets, agent_path):
# Creates priority queue based on
    target_paths = []
# step length
    for target in targets:
        target_paths.append(
            path_merge(
# Create paths from
                agent_path,
# current agent location
                targets[f'{target}'].root_path,
# and targets
                method='path')
        )
    target_paths.sort(key=len)
# Sort ascending by length and return
    return target_paths
```