Data Structures & Algorithms (Goodarzi/Dimmery) Problem Set 2

Paul Elvis Otto | 249968

2025-03-23

Task 1

Solved under the follwoing domain: https://github.com/Paul-Elvis-Otto/calculator_app_dsa_assignement

Task 2.a)

The `Agent` class is an abstract base class that defines the interface for `Proposer` and `Proposee` subclasses. `Proposer` implements `propose()` by selecting a preferred proposee from its list, while raising an error for `respond()`. Conversely, `Proposee` implements `respond()` with preference-based matching logic, while raising an error for `propose()`. This models a one-way matching system like the stable marriage problem.

Task 2.b)

```python
from abc import ABC, abstractmethod
from typing import List, Dict, Optional
class Agent(ABC):
    def __init__(self, name: str, preferences: list[str]):
        self.name = name
        self.preferences = preferences
        self.match = None

    @abstractmethod
    def propose(self):
        pass
    @abstractmethod
    def respond(self, proposer: 'Agent') -> bool:
        pass

class Proposer(Agent):
    def propose(self):
        """
        Propose to the next preferred proposee.
        """
        if not self.preferences:
            raise ValueError(f"{self.name} has no more preferences to propose to.")
        return self.preferences.pop(0)

    def respond(self, proposer: 'Agent') -> bool:
        """
        Proposers do not respond to proposals; they only propose.
        """
        raise NotImplementedError("Proposers cannot respond to proposals.")

class Proposee(Agent):
    def propose(self):
        """
        Proposees do not propose; they only respond.
        """
        raise NotImplementedError("Proposees cannot propose.")

    def respond(self, proposer: 'Agent') -> bool:
        """
        Respond to a proposal based on preferences.
        """
        if self.match is None:
            return True
        else:
            # If proposee is matched, compare preferences
            current_match_index = self.preferences.index(self.match.name)
            proposer_index = self.preferences.index(proposer.name)
            if proposer_index < current_match_index:
                return True
            else:
                return False

# Gale-Shapley Algorithm
```

```python
class GaleShapley:
    def __init__(self, proposers: list[Proposer], proposees: list[Proposee]):
        self.proposers = proposers
        self.proposees = proposees
        self.matches = {}  # Stores the final matches

    def match(self) -> dict:
        """
        Run the Gale-Shapley algorithm to find a stable matching.
        """
        free_proposers = list(self.proposers)  # Initialize all proposers as free

        while free_proposers:
            proposer = free_proposers.pop(0)  # Pick a free proposer
            proposee_name = proposer.propose()  # Propose to the next preferred proposee
            proposee = next(p for p in self.proposees if p.name == proposee_name)

            # YOUR CODE HERE
            # If the proposee accepts the proposal
            if proposee.respond(proposer):
                # If proposee was already matched, free the previous match
                if proposee.match is not None:
                    previous_match = proposee.match
                    previous_match.match = None
                    free_proposers.append(previous_match)

                # Match the proposer and proposee
                proposer.match = proposee
                proposee.match = proposer
                self.matches[proposer.name] = proposee.name
            else:
                # If rejected, put proposer back in the free list
                free_proposers.append(proposer)

        return self.matches

# Example from slide #13 (assuming)
# Test Case 1: Men as proposers
def test_case_1():
    print("Test Case 1: Men as proposers")

    # Create proposers (men)
    men = [
        Proposer("A", ["X", "Y", "Z"]),
        Proposer("B", ["Y", "Z", "X"]),
        Proposer("C", ["Y", "X", "Z"])
    ]

    # Create proposees (women)
    women = [
        Proposee("X", ["B", "A", "C"]),
        Proposee("Y", ["C", "B", "A"]),
        Proposee("Z", ["A", "C", "B"])
    ]

    # Run Gale-Shapley algorithm
    gs = GaleShapley(men, women)
    matches = gs.match()

    print("Final matches:")
    for proposer, proposee in matches.items():
        print(f"{proposer} → {proposee}")

    # Expected: A-Z, B-X, C-Y (based on standard Gale-Shapley with men proposing)
    print("\n")

# Test Case 2: Women as proposers
def test_case_2():
    print("Test Case 2: Women as proposers")

    # Create proposers (women)
    women_proposers = [
        Proposer("X", ["B", "A", "C"]),
        Proposer("Y", ["C", "B", "A"]),
        Proposer("Z", ["A", "C", "B"])
    ]

    # Create proposees (men)
    men_proposees = [
        Proposee("A", ["X", "Y", "Z"]),
        Proposee("B", ["Y", "Z", "X"]),
        Proposee("C", ["Y", "X", "Z"])
    ]
    # Run Gale-Shapley algorithm
    gs = GaleShapley(women_proposers, men_proposees)
    matches = gs.match()

    print("Final matches:")
    for proposer, proposee in matches.items():
        print(f"{proposer} → {proposee}")

    # Expected outcome when women propose

# Run the tests
print("Testing Gale-Shapley Algorithm Implementation")
print("-" * 60)
test_case_1()
print("-" * 60)
test_case_2()
```

```
Testing Gale-Shapley Algorithm Implementation
------------------------------------------------------------
Test Case 1: Men as proposers
Final matches:
A → X
B → Z
C → Y


------------------------------------------------------------
Test Case 2: Women as proposers
Final matches:
X → B
Y → C
Z → A
```

Task 3

Task 3.a)

The algorithm is not efficient. Its time complexity is $O(n^2)$ where n represents the size of the array. The algorithm systematically examines all possible contiguous subarrays by using two nested loops. The outer loop iterates through each starting position in the array, while the inner loop extends the subarray from that position to all possible ending positions. For each subarray configuration, the algorithm calculates the sum and updates the maximum if necessary. This approach requires examining approximately $n^2/2$ subarrays, which simplifies to $O(n^2)$ in Big-O notation. Therefore, the smallest value of d such that O(n^d) correctly represents the runtime complexity is d = 2.

Task 3.b)

The second algorithm implements Kadane's algorithm for finding the maximum subarray sum. It operates with a time complexity of O(n). This algorithm processes the array in a single linear pass, maintaining two variables: a running sum of the current subarray and the maximum sum found so far. For each element, the algorithm makes a critical decision: either start a new subarray beginning with the current element (if the previous running sum became negative) or extend the existing subarray by adding the current element. This approach leverages dynamic programming principles with optimal substructure.

The difference in complexity between the two algorithms is substantial. While the first algorithm examines all possible subarrays through brute force requiring quadratic time, the second algorithm achieves the same result in linear time by avoiding redundant calculations. This efficiency difference becomes particularly significant for large inputs. For an array of 10,000 elements, the first algorithm would perform approximately 50 million operations, whereas the second would require only 10,000 operations. The second algorithm demonstrates how intelligent algorithm design can dramatically reduce computational requirements while solving the same problem.