A Cryptocurrency Explorer & Market Analyzer

Janhvi Goje

Function 1: `read_file()`

```
def read_file(file_path: str) -> any:
    data = []
    with open(file_path, 'r') as file:
        for line in file:
            split_line = line.split(",", 4)
            split_line[0]=split_line[0].lower()
            split_line[1]=int(split_line[1])
            split_line[2]=float(split_line[2])
            split_line[3]=float(split_line[3])
            data.append(split_line)
    return data
```

- We don't use file.readlines() as it reads all the lines in the file into memory at once.
- For dataset_full.txt, this can potentially use up a lot of memory.
- Instead, the file is read line by line, which is more memory-efficient.

- `open(file_path, 'r') as file` statement ensures that the file is properly closed after it is no longer needed.
- This is a good practice as it prevents resource leaks, avoiding potential problems that can arise from leaving files open.

- crypto_names are read with lower() to avoid any problems that occur with ASCII code allocation that sorts **small z** first and then **capital A.**
- Further useful in sort_data()

Function 2: `crypto_stat()`

```
def crypto_stats(data, crypto_name: str, interval: Tuple[int, int]) -> Tuple[float,
float, float]:
    sum, count = 0, 0
    mn, mx, avg = float('inf'), 0.0, 0.0
    for var in data :
        if var[0] == crypto_name.lower() and interval[0] <= var[1] <= interval[1]:
            try:
                if mn > var[2]:
                    mn = var[2]
                if mx < var[2]:
                    mx = var[2]
                sum = sum + var[2]
                count += 1
            except KeyError:
                return (0.0, 0.0, 0.0)
    avg = sum/count
    return (round(mn, 2),round(avg, 2), round(mx, 2))
```

- `float('inf')` used instead of 99999 as per feedback.

- The function iterates over the data list once, and for each iteration, it performs a constant number of operations (checking conditions, potentially updating variables). Therefore, **the overall time complexity is linear, O(n).**
- O(n) is most optimal for this problem as **we must examine each element in the list at least once to determine if it fits within the specified interval** and if it should be considered for the minimum, maximum, and average calculations

- The **space complexity of this code is O(1),** which means the amount of memory used does not change with the size of the input data set.

Function 3: `sort_data()`

```
def sort_data(data) -> List[Tuple[str, float]]:

    def quick_sort(data):
        elements = len(data)
        if elements < 2:
            return data
        pivot = 0
        for i in range(1, elements):
            if data[i] <= data[0]:
                pivot += 1
                data[i], data[pivot] = data[pivot], data[i]

        data[0], data[pivot] = data[pivot], data[0]
        low = quick_sort(data[0:pivot])
        high = quick_sort(data[pivot+1:elements])
        data = low + [data[pivot]] + high
        return data

    def sorted_data(data):
        data = quick_sort(data)
        result = []
        for i in range(0, len(data)):
            crypto_and_price = (data[i][0], data[i][1])
            result.append(crypto_and_price)
        return result

    sorted = sorted_data(data)
    return sorted
```

- **Uses First-Element Pivot Selection** to randomise the selection and keep the time complexity mostly limited to the average case of **O(nlogn)**.
- The **sorted_data** function simply iterates over the sorted list once to form the tuples, so its **time complexity is O(n)**.
- It's important to note that, in practice, **quicksort is often faster** than other O(n log n) algorithms like mergesort or heapsort, due to its in-place characteristic and good cache performance.

- It is noteworthy that the **worst-case scenario of quick_sort (O(n^2)) would only occur if the list is already sorted which is highly unlikely** as we are dealing unsorted datasets.

Function 4: `get_max_value()`

```
def get_max_value(data, crypto: str, month: int) -> Tuple[int, float]:
    start_day = (month - 1) * 30 + 1
    end_day = month * 30

    max_value = (-1, -1)

    for item in data:
        if item[0] == crypto.lower() and start_day <= item[1] <= end_day:
                if item[2] > max_value[1]:
                    max_value = (item[1], item[2])

    return max_value
```

- The function has a **linear time complexity (O(n))**, which is the best achievable for this type of problem because in the worst case, we have to inspect every item in the list to ensure you've found the maximum price for a specific cryptocurrency in a specific month.
- As for space complexity, the function is optimal with a **space complexity of O(1)** because it only uses a constant amount of additional space that doesn't grow with the size of the input.

Function 5: `search()`

```
def search(data, value: float, crypto: str) -> Tuple[int, float]:

    price = 0
    day = 0

    for item in data:
        if item[0] == crypto.lower():
            price_difference = abs(item[2] - value)
            if price_difference == 0:
                return (item[1], item[2])
            elif price_difference < abs(price - value):
                price = item[2]
                day = item[1]

    return (int(day), price)
```

- The **time complexity of this function is O(n)** because in the worst-case scenario, you have to inspect every item in the list to ensure you've found the price closest to the desired value for a specific cryptocurrency.
- As for **space complexity, the function is O(1),** which means it uses a constant amount of space. The function only needs a fixed number of variables (price, day) to store intermediate results, and these variables are updated in-place during the iteration. The function doesn't create any new data structures that grow with the size of the input data.

Function 6: `min_correlation_pathways()`

```
def min_correlation_pathways(data: List[Tuple[str, int, float, float]],
                             crypto: str,
                             interval: Tuple[int, int]) -> Dict[str, List[Tuple[str, float]]]:
    crypto=crypto.lower()
    . . . make the graph as a dictionary of dictionaries . . . #refer the code

    def prim(graph, start_node):
        mst = {node: [] for node in graph}
        visited = set([start_node])
        edges = [
            (cost, start_node, to)
            for to, cost in graph[start_node].items()
        ]
        heapq.heapify(edges)

        while edges:
            cost, frm, to = heapq.heappop(edges)
            if to not in visited:
                visited.add(to)
                mst[frm].append((to, cost))
                for to_next, cost2 in graph[to].items():
                    if to_next not in visited:
                        heapq.heappush(edges, (cost2, to, to_next))
        return mst
    mst = prim(graph, crypto)
    return mst
```

- This function uses **Prim's Algorithm for MST** which is highly appropriate as this algorithm always starts with a single node and moves through the adjacent nodes, in order to explore all of the connected edges along the way.
- An alternate take on this was the *Kruskal's Algorithm*, which wasn't used as it **isn't the best choice for large datasets like dataset_full.txt.** Prim's algorithm is more efficient as it depends on the **priority queue implementation of the number of edges** and in our dense graph, the number of edges is relatively high, so the overall number of iterations and edge comparisons is significant.

Function 7: `correlated_cryptos_at_lvl_k()`

```python
def correlated_cryptos_at_lvl_k(data: List[Tuple[str, int, float, float]],
                                crypto: str,
                                level: int,
                                interval: Tuple[int,int]) -> List[str]:
    crypto=crypto.lower()
    def bfs_levels(mst, start_node):
        levels = {node: None for node in mst}
        levels[start_node] = 0
        queue = [start_node]

        while queue:
            node = queue.pop(0)
            for neighbour, _ in mst[node]:
                if levels[neighbour] is None:
                    levels[neighbour] = levels[node] + 1
                    queue.append(neighbour)
        return levels

    def get_nodes_at_level_k(mst, start_node, k):
        levels = bfs_levels(mst, start_node)
        return [node for node, level in levels.items() if level == k]
    mst = min_correlation_pathways(data, crypto, interval)
    level_k_nodes = get_nodes_at_level_k(mst, crypto, level)
    return level_k_nodes
```

- Using **Breadth-First Search (BST)** to traverse through the MST generated by the Prim's Algorithm.
- Here **bfs_levels(mst, start_node)** function is implemented to traverse the MST using BFS and **determine the level** of each node. The **get_nodes_at_level_k(mst, start_node, k)** function then **uses this obtained level to get all nodes at level k**.