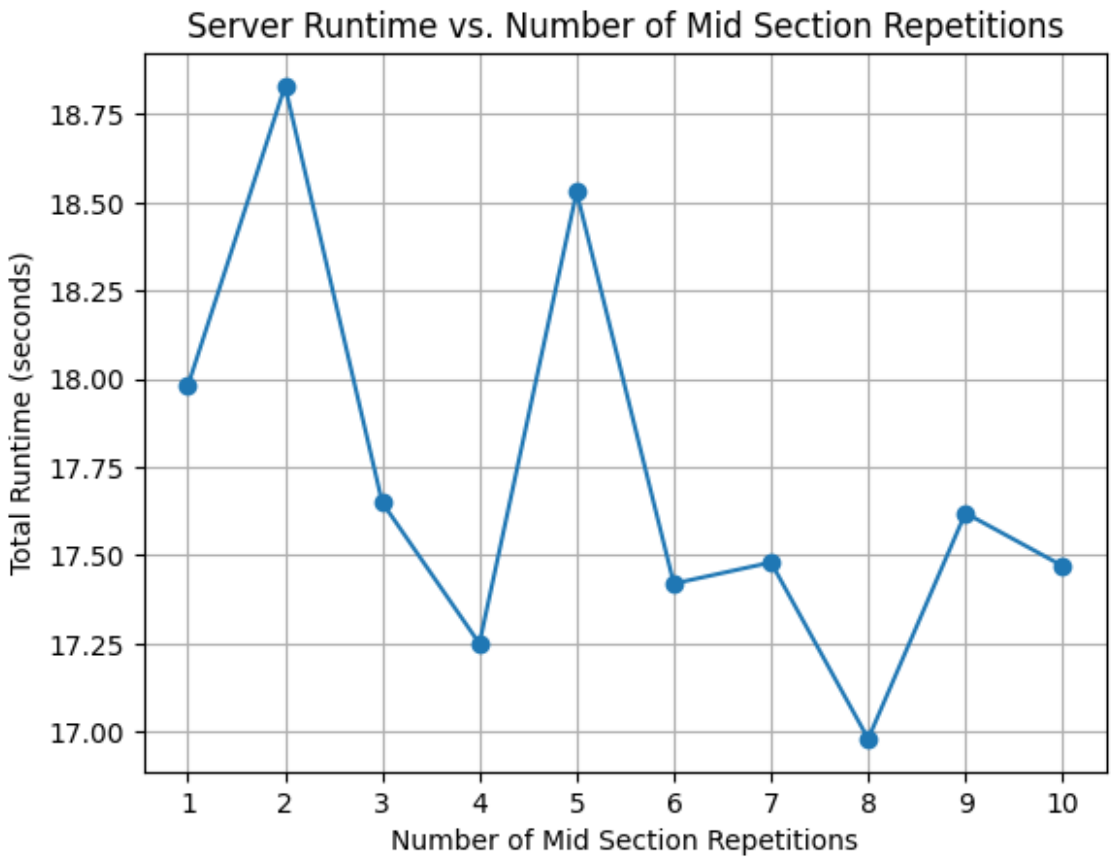
Part A:

Graph below.

```
import matplotlib.pyplot as plt

# Elapsed times for all runs in minutes and seconds
elapsed_times = [
    "0:17.98", "0:18.83", "0:17.65", "0:17.25",
    "0:18.53", "0:17.42", "0:17.48", "0:16.98",
    "0:17.62", "0:17.47"
]

# Convert elapsed times to seconds
runtimes = []
for elapsed in elapsed_times:
    minutes, seconds = map(float, elapsed.split(':'))
    total_seconds = minutes * 60 + seconds
    runtimes.append(total_seconds)

# Number of Mid Section Repetitions for each run
repetitions = list(range(1, len(runtimes) + 1))

# Plot the results
plt.plot(repetitions, runtimes, marker='o')
plt.title('Server Runtime vs. Number of Mid Section Repetitions')
plt.xlabel('Number of Mid Section Repetitions')
plt.ylabel('Total Runtime (seconds)')
plt.xticks(repetitions)  # Ensure x-axis has correct number of
repetitions
plt.grid(True)
plt.show()
```

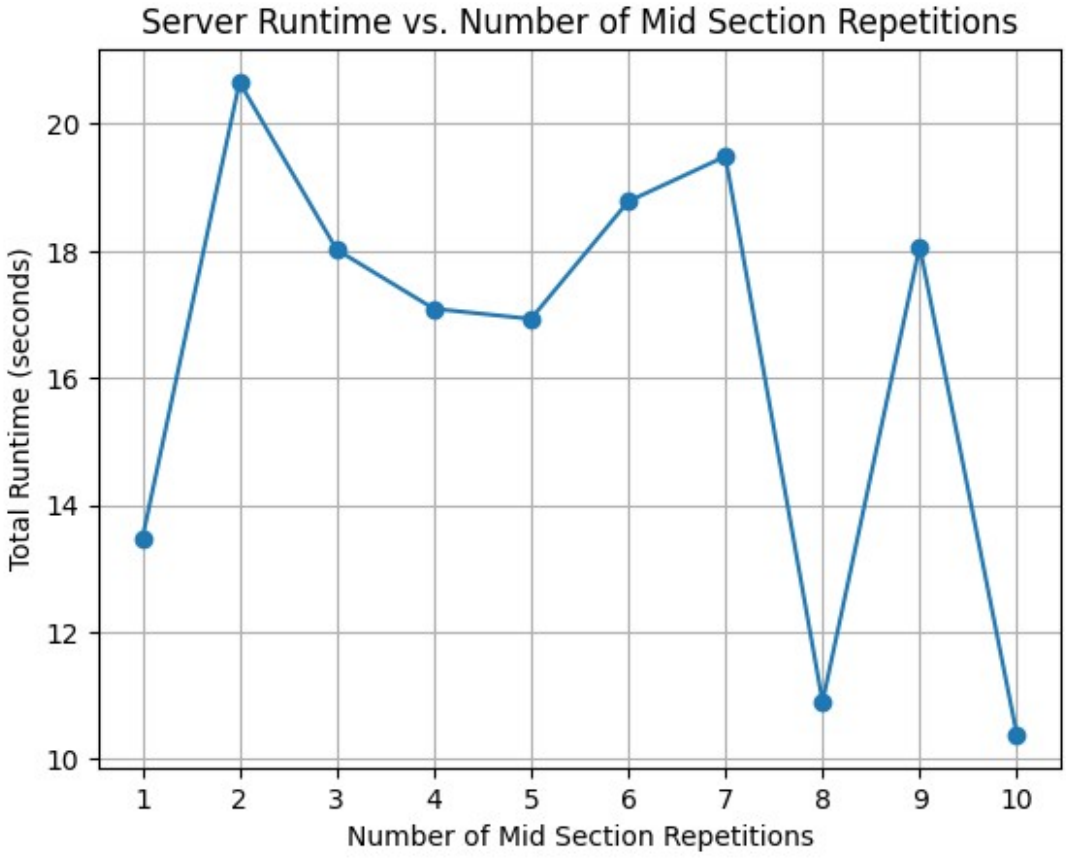


Part B:

Graph below. There was definitely a noticable speedup with the use of multi-threading. However, the speedup was not as significant as I thought it would be. For the most part, the times were still in the same ballpark as with the single-threaded experiment.

```
import matplotlib.pyplot as plt

# Elapsed times for all runs in minutes and seconds
elapsed_times = [
    "0:13.47", "0:20.66", "0:18.02", "0:17.09",
    "0:16.93", "0:18.78", "0:19.49", "0:10.90",
    "0:18.07", "0:10.38"
]

# Convert elapsed times to seconds
runtimes = []
for elapsed in elapsed_times:
    minutes, seconds = map(float, elapsed.split(':'))
    total_seconds = minutes * 60 + seconds
    runtimes.append(total_seconds)

# Number of Mid Section Repetitions for each run
```

```
repetitions = list(range(1, len(runtimes) + 1))

# Plot the results
plt.plot(repetitions, runtimes, marker='o')
plt.title('Server Runtime vs. Number of Mid Section Repetitions')
plt.xlabel('Number of Mid Section Repetitions')
plt.ylabel('Total Runtime (seconds)')
plt.xticks(repetitions)  # Ensure x-axis has correct number of
repetitions
plt.grid(True)
plt.show()
```

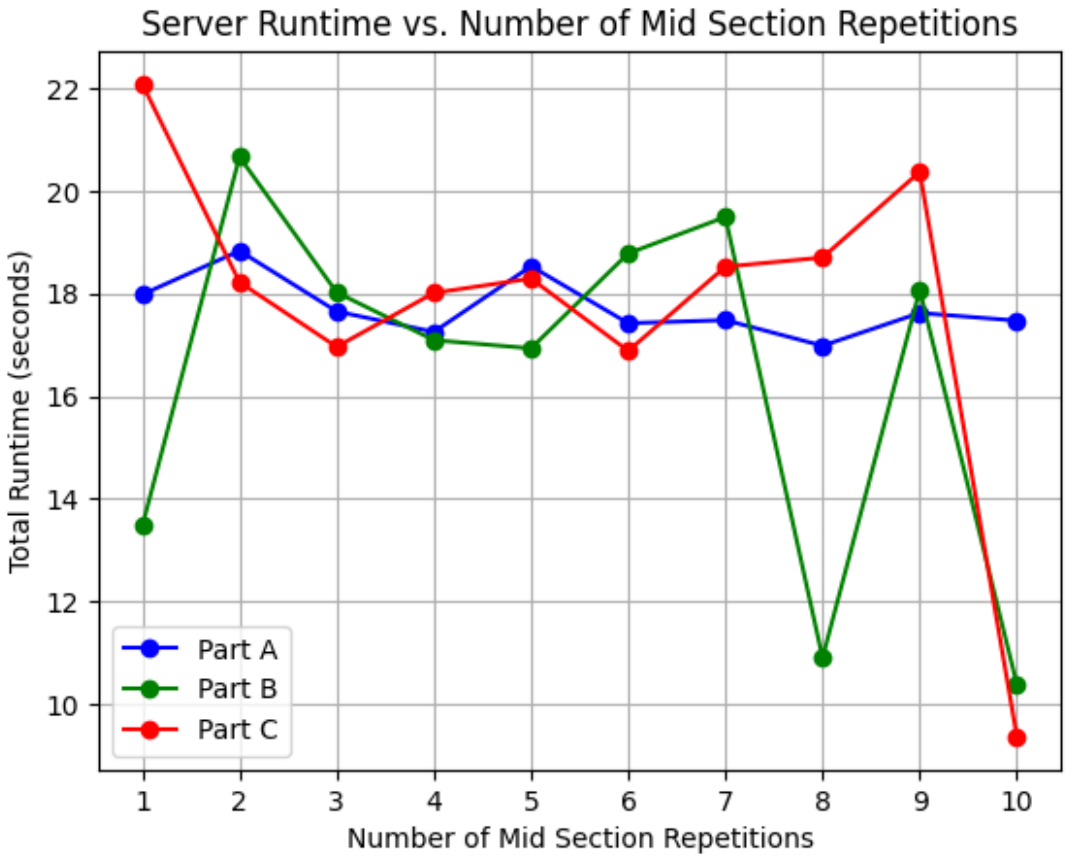


Part C:

Graph below. The question asks about an additional speedup that I should be seeing from part B to part C, but I am not exactly seeing a sizable one if any. Part C was faster than Part B on exactly 5 out of the 10 runs, so I would say that if there was a speed up that I should be expecting, I did not get it. It is probably because my machine is very old and feeble. I apologize about that. I see that the same exact operations are being performed in both B and C, perhaps the order has an impact on the server's performance.

```
import matplotlib.pyplot as plt
```

```
def convert_to_seconds(elapsed_times):
    runtimes = []
    for elapsed in elapsed_times:
        minutes, seconds = map(float, elapsed.split(':'))
        total_seconds = minutes * 60 + seconds
        runtimes.append(total_seconds)
    return runtimes

# Elapsed times for all runs in minutes and seconds
elapsed_times_parta = [
    "0:17.98", "0:18.83", "0:17.65", "0:17.25",
    "0:18.53", "0:17.42", "0:17.48", "0:16.98",
    "0:17.62", "0:17.47"
]
elapsed_times_partb = [
    "0:13.47", "0:20.66", "0:18.02", "0:17.09",
    "0:16.93", "0:18.78", "0:19.49", "0:10.90",
    "0:18.07", "0:10.38"
]
elapsed_times_partc = [
    "0:22.08", "0:18.22", "0:16.96", "0:18.01",
    "0:18.29", "0:16.88", "0:18.52", "0:18.70",
    "0:20.37", "0:09.36"
]

# Convert elapsed times to seconds
runtimes_parta = convert_to_seconds(elapsed_times_parta)
runtimes_partb = convert_to_seconds(elapsed_times_partb)
runtimes_partc = convert_to_seconds(elapsed_times_partc)

# Number of Mid Section Repetitions for each run
repetitions = list(range(1, 11))

# Plot the results for each part
plt.plot(repetitions, runtimes_parta, marker='o', color='blue',
label='Part A')
plt.plot(repetitions, runtimes_partb, marker='o', color='green',
label='Part B')
plt.plot(repetitions, runtimes_partc, marker='o', color='red',
label='Part C')

plt.title('Server Runtime vs. Number of Mid Section Repetitions')
plt.xlabel('Number of Mid Section Repetitions')
plt.ylabel('Total Runtime (seconds)')
plt.xticks(repetitions)
plt.legend()  # Add a legend to distinguish the parts
plt.grid(True)
plt.show()
```



Part D:

Constructing a Scheduling Policy:

Group incoming requests by the type of image operation requested. Have separate queues for each type of operation. Schedule operations from these queues using round-robin, so it we can be sure that a batch of similar operations is executed consecutively before moving to the next type.

Within each operation type, use SJN to decide which requests are to be serviced. Apply a dynamic adjustment mechanism that gives priority to operation types that are falling behind to ensure all types progress evenly. The rationale behind this policy is that by executing similar operations consecutively, the server can take advantage of potential optimizations like instruction and data locality, leading to performance gains that are consistent regardless of the initial request order.

To implement this, the server would need a scheduler that can classify incoming requests, distribute them into the correct queues, and select the next batch of operations to execute based on the scheduling logic. The scheduler would also monitor the completion of tasks and adjust priorities to maintain the flow of operations.