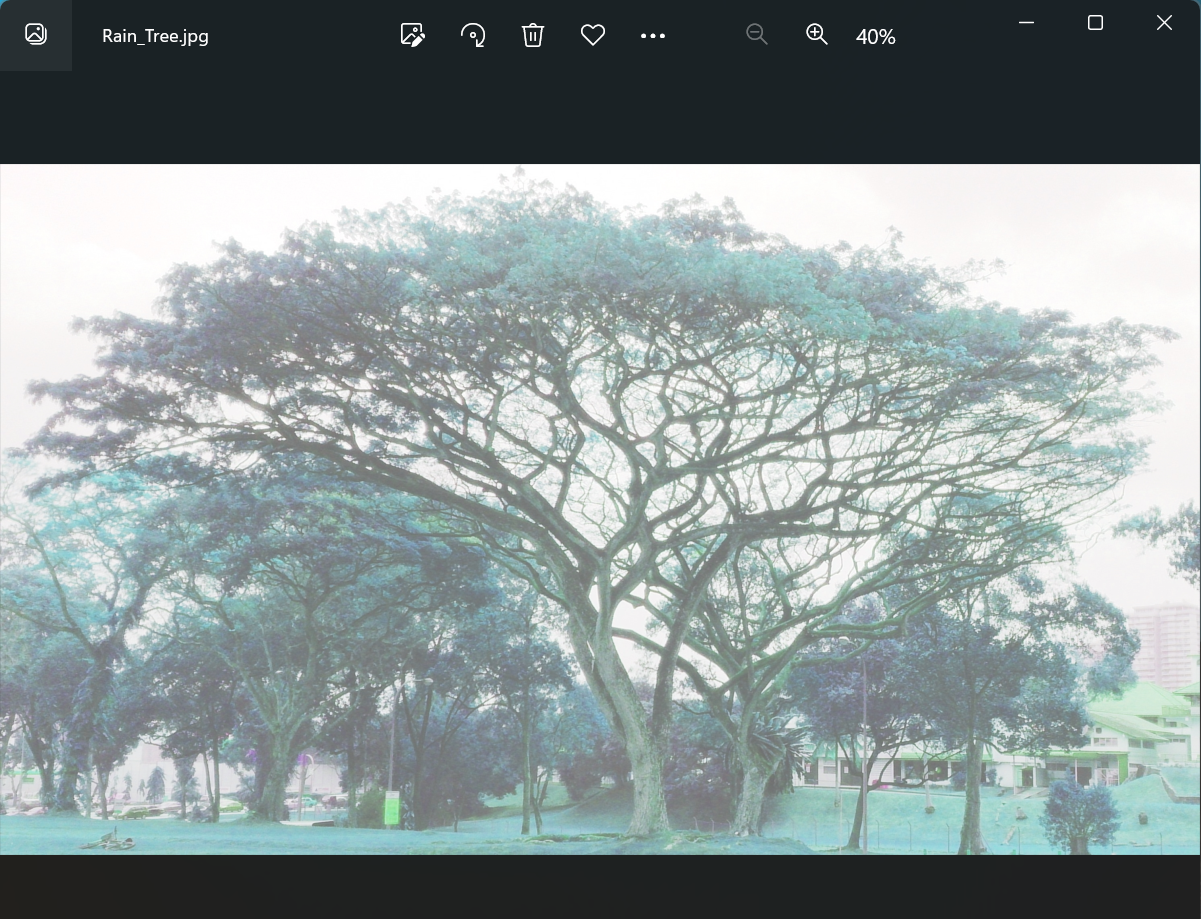
PROBLEM 1

# INPUT IMAGE:



# OUTPUT IMAGES:

Single-threaded image   
A screenshot of a black and white photo of a tree

Description automatically generated with low confidence

## Multi-threaded Image

A screenshot of a black and white image of a tree

Description automatically generated with low confidence

# EXPLANATION:

This code is an implementation of image histogram equalization using both single-threaded and multi-threaded approaches.

1. The code begins with the package declaration and imports necessary classes and libraries.
2. The ‘combined’ class is declared as the main class for the program.
3. ‘numOfThreads’ is a constant variable representing the number of threads to be used for multi-threaded processing.
4. The `main` method is the entry point of the program. It performs the following steps:
   * Records the start time using ‘System.currentTimeMillis()’.
   * Calls the ‘singleThreadedEqualize()’ method to perform histogram equalization using a single thread.
   * Records the end time and calculates the execution time (‘singleTime’).
   * Prints the execution time for the single-threaded approach.
   * Records the start time again.
   * Calls the ‘multiThreadedEqualize()’ method to perform histogram equalization using multiple threads.
   * Records the end time and calculates the execution time (‘multiTime’).
   * Prints the execution time for the multi-threaded approach.
5. The ‘singleThreadedEqualize()’ method performs the histogram equalization using a single thread. It does the following:
   * Loads an image from the specified file path.
   * Converts the image to grayscale using the ‘Grayscale()’ method.
   * Calculates the histogram of the grayscale image using the `calHistogram()` method.
   * Calculates the cumulative histogram using the `calCumulative()` method.
   * Performs the equalization of the image using the `equalizeImage()` method.
   * Saves the equalized image to the specified output file path using the `saveImage()` method.
6. The `multiThreadedEqualize()` method performs the histogram equalization using multiple threads. It does the following:
   * Loads an image from the specified file path.
   * Converts the image to grayscale using the `Grayscale()` method.
   * Divides the image into equal parts for each thread based on the specified number of threads (`numOfThreads`).
   * Creates an array of threads and starts them.
   * Each thread processes its assigned part of the image by iterating over the pixels and converting them to grayscale.
   * Waits for all threads to finish using the `join()` method.
   * Calculates the histogram of the grayscale image using the `calHistogram()` method.
   * Calculates the cumulative histogram using the `calCumulative()` method.
   * Performs the equalization of the image using the `equalizeImage()` method.
   * Saves the equalized image to the specified output file path using the `saveImage()` method.
7. The `Grayscale()` method takes a `BufferedImage` as input and converts it to grayscale. It does the following:
   * Creates a new `BufferedImage` object with the same dimensions as the input image.
   * Iterates over each pixel in the image.
   * Retrieves the RGB values of each pixel and calculates the average to obtain the grayscale value.
   * Constructs a new grayscale pixel value by combining the grayscale value three times (for red, green, and blue components).
   * Sets the new grayscale pixel value in the corresponding position of the grayscale image.
   * Returns the grayscale image.
8. The `calHistogram()` method takes a grayscale image as input and calculates the histogram. It does the following:
   * Retrieves the width and height of the image.
   * Creates an array `histogram` with 256 elements to store the histogram.
   * Iterates over each pixel in the image.
   * Extracts the grayscale value of each pixel and increments the corresponding histogram bin.
   * Returns the histogram array.
9. The `calCumulative()` method takes a histogram array as input and calculates the cumulative histogram. It does the following:
   * Creates an array `cumulativeHist` with 256 elements to store the cumulative histogram.
   * Sets the first element of `cumulativeHist` to the value of the first element in the input histogram.
   * Iterates over the remaining elements of the histogram array.
   * Calculates each cumulative histogram bin by adding the previous cumulative value and the current histogram value.
   * Returns the cumulative histogram array.
10. The `equalizeImage()` method takes a grayscale image and a cumulative histogram as input and performs histogram equalization. It does the following:
    * Retrieves the width and height of the image.
    * Calculates the total number of pixels in the image.
    * Iterates over each pixel in the image.
    * Retrieves the grayscale value of each pixel.
    * Calculates the equalized grayscale value by multiplying the cumulative histogram value by 255 and dividing by the total number of pixels.
    * Constructs a new RGB value using the equalized grayscale value.
    * Sets the new RGB value in the corresponding position of the image.
11. The `saveImage()` method takes a `BufferedImage` and a file path as input and saves the image to the specified output path. It does this by using the `ImageIO.write()` method.

Overall, this code demonstrates how to perform histogram equalization on an image using both single-threaded and multi-threaded approaches. The single-threaded method does these stages in a sequential order, but the multi-threaded approach divides the image into equal parts and processes them concurrently using multiple threads to potentially improve the performance.

# TIMING ANALYSIS:

For fair comparison, when obtaining the execution time for single thread, the multithread part of the program was disabled (commented). For each number of threads, three tests were taken and used the average of the three for the graph.

**numOfThreads = 4**

1st test



2nd test

A screenshot of a computer

Description automatically generated with medium confidence

3rd test

A picture containing text, font, screenshot, line

Description automatically generated

**numOfThreads = 3**

1st test



2nd test

A picture containing text, font, screenshot, line

Description automatically generated

3rd test

A picture containing text, font, screenshot, line

Description automatically generated

**numOfThreads = 2**

1st test

A picture containing text, font, screenshot, line

Description automatically generated

2nd test

A picture containing text, font, screenshot, line

Description automatically generated

3rd test



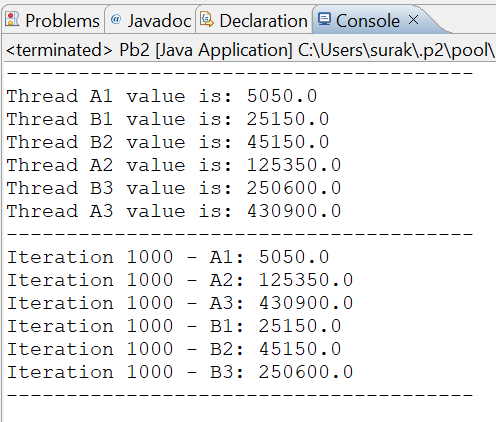
Based on the results, it is evident that the multi-threaded execution time is lesser than the single-threaded execution time. As the number of threads is reduced, the difference between the execution times of multi-threaded and single-threaded approaches diminishes. Additionally, as the number of threads decreases, the single-threaded execution time decreases slightly, while the multi-threaded execution time experiences a minor increase.

|  |  |  |  |
| --- | --- | --- | --- |
| No. of Threads | 4 | 3 | 2 |
| Single thread average | 1380 | 1301 | 1210.333 |
| Multi thread average | 1098.333 | 1099.333 | 1115.333 |
| Difference time | 281.6667 | 201.6667 | 95 |

Therefore, we can see that the multi-threaded execution time is faster than single-threaded execution time.

PROBLEM 2

# OUTPUT SNIPPET:



# QUESTION:

**Active wait (i.e. while(!cond){} which wastes CPU cycles ) or Thread sleep method should not be used in your solution. Explain in detail your answer.**

Active waiting and the 'Thread.sleep()' function are not utilised for synchronisation in the code. Instead, the code makes use of the 'wait()' and 'notify()' functions, which have boolean flags. Active waiting, which constantly checks a condition in a loop, is wasteful because it uses CPU cycles while doing nothing beneficial. Threads can stop execution with the 'wait()' function, freeing up system resources until they get notification from another thread. Threads are woken only when relevant circumstances are satisfied by utilising 'wait()' and 'notify()', conserving CPU cycles and preventing excessive resource use. The code also eliminates arbitrary sleep intervals, assuring synchronisation based on the completion of specified calculations rather than preset delays. This method enhances thread synchronisation efficiency.

# Implementation of Synchronization:

The code represents a multi-threaded program with two threads that perform calculations and share data. The program implements synchronization using the `synchronized` keyword, `wait()`, and `notify()` methods. Synchronization ensures that only one thread can access shared resources at a time, preventing data corruption or race conditions.

1. Shared Data and Flags:
   1. The `Data` class contains shared variables and boolean flags. The boolean flags (`GoA1`, `GoA2`, `GoA3`, `GoB1`, `GoB2`, `GoB3`) act as synchronization signals between threads to indicate when specific calculations are ready to be performed. The shared variables (`A1`, `A2`, `A3`, `B1`, `B2`, `B3`) hold the calculated values.
2. Synchronized Method:
3. The `Sum()` method in the `Data` class is declared as `synchronized`. This ensures that only one thread can execute this method at a time, preventing concurrent modifications of the shared variable `s`.
4. Thread1:

In the `run()` method of `Thread1` class, the code follows the following synchronization steps:

* For function A1:
  + The code enters a synchronized block using `synchronized (d)`.
  + It calculates the value of A1 using `Data.Sum(100)`.
  + Sets the flag `GoB1` to `true` to signal Thread2 that A1 value is ready.
  + Prints the calculated value of A1.
  + Calls `notify()` to notify other threads waiting on the same object (`d`).
* For function A2:
* The code enters another synchronized block.
* It waits using `wait()` until the `GoA2` flag becomes `true`.
* Once the flag is set, it calculates the value of A2 using `d.B2 + Data.Sum(400)`.
* Sets the flag `GoB3` to `true` to signal Thread2 that A2 value is ready.
* Calls `notify()` to notify other threads waiting on the same object (`d`).
* For function A3:
* The code follows a similar pattern as function A2.
* It waits using `wait()` until the `GoA3` flag becomes `true`.
* Once the flag is set, it calculates the value of A3 using `d.B3 + Data.Sum(600)`.
* Calls `notify()` to notify other threads waiting on the same object (`d`).

1. Thread2:

In the `run()` method of `Thread2` class, the code follows the following synchronization steps:

* For function B1:
* The code enters a synchronized block.
* It waits using `wait()` until the `GoB1` flag becomes `true`.
* Once the flag is set, it calculates the value of B1 using `d.A1 + Data.Sum(200)`.
* Sets the flag `GoB2` to `true` to signal Thread1 that B1 value is ready.
* Calls `notify()` to notify other threads waiting on the same object (`d`).
* For function B2:
* The code enters another synchronized block.
* It waits using `wait()` until the `GoB2` flag becomes `true`.
* Once the flag is set, it calculates the value of B2 using `Data.Sum(300)`.
* Sets the flag `GoA2` to `true` to signal Thread1 that B2 value is ready.
* Calls `notify()` to notify other threads waiting on the same object (`d`).
* For function B3:
* The code follows a similar pattern as function B2.
* It waits using `wait()` until the `GoB3` flag becomes `true`.
* Once the flag is set, it calculates the value of B3 using `d.A2 + Data.Sum(500)`.
* Sets the flag `GoA3` to `true` to signal Thread1 that B3 value is ready.
* Calls `notify()` to notify other threads waiting on the same object (`d`).

To execute numerous iterations, the code inserts a loop in the ‘main()’ function. Each iteration generates a new instance of 'Data' and launches two threads (‘Thread1’ and ‘Thread2’). After both threads have completed their execution, the estimated values of A1, A2, A3, B1, B2, and B3 for the current iteration are reported.

Synchronisation guarantees that threads execute in the correct sequence, waiting for the appropriate signals before conducting their calculations. The shared variables are accessed and changed synchronously to avoid data races or inconsistencies.

# GITHUB LINK:

<https://github.com/suraksha1313/331codes>