KARAN DOSHI
DRAGONFRUIT AI CHALLENGE
GITHUB LINK: https://github.com/doshikaran/Dragonfruit-AI-Challenge.git

<u>QUESTION 01:</u>
Come up with efficient data structures to represent both types of images: those generated by the microscope, and those generated by the dye sensor. These need not have the same representation; the only requirement is that they be compact and take as little storage space as possible. Explain why you picked the representation you did for each image type, and if possible, estimate how much storage would be taken by the images. What is the worst-case storage size in bytes for each image representation you chose?

<u>ANSWER 01</u>:
The run-length encoding (RLE) or quadtree representations would be best for microscope images because they can effectively compress the binary image data where the parasite body is represented as a blob of black pixels against a white background.

RLE is a simple and effective compression technique for binary or simple images with long runs of consecutive pixels of the same color. It works by encoding consecutive pixels of the same color as a single value and a count, instead of storing each pixel individually. This can lead to significant space savings, especially for images with large contiguous regions of the same color, such as the microscope images with a solid blob.

Similarly, a **quadtree** is a tree data structure that recursively subdivides the image into four equal quadrants or regions. If a region is homogeneous (all pixels have the same color), it is represented as a leaf node with that color value. If a region is not homogeneous, it is further subdivided into four quadrants. This process continues until all regions are homogeneous or a specified maximum depth is reached. Quadtrees can effectively represent binary images with large homogeneous regions, like the microscope images with a solid blob.

While these two representations are suitable for the microscope images, they are not optimal for the dye sensor images, where the dye can be present both inside and outside the parasite body. In such cases, the dye distribution may be more scattered and irregular, making run-length encoding or quadtrees less effective.

Therefore, for the dye sensor images, a **sparse matrix** representation is a good choice. In a sparse matrix, only the non-zero elements are stored, along with their row and column indices. This can be an efficient way to represent the locations of lit pixels (non-zero elements) in the dye sensor image, as it is expected that the number of lit pixels will be relatively small compared to the total number of pixels in the image.

To analyze the worst-case storage size for the proposed image representations, let's consider the given image dimensions of 100,000 x 100,000 pixels.
Run-Length Encoding (RLE) for Microscope Images:

- In the worst case, the parasite blob could be a single horizontal or vertical line, where every pixel is different from its neighbors.
- For a 100,000 x 100,000 image, the worst case would be a run of length 1 for all 10^10 pixels.
- In RLE, each run requires storing the color value (1 byte) and the run length (4 bytes for a 32-bit integer).
- Therefore, the worst-case storage size for RLE would be 10^10 * (1 byte + 4 bytes) = 50 GB.

Sparse Matrix for Dye Sensor Images

- In the worst case, every pixel in the 100,000 x 100,000 image could be lit (non-zero).
- For a sparse matrix representation, each non-zero element requires storing the row index (4 bytes for a 32-bit integer), column index (4 bytes), and the value (1 byte, assuming a binary value indicating lit or unlit).
- Therefore, the worst-case storage size for the sparse matrix would be 10^10 * (4 bytes + 4 bytes + 1 byte) = 90 GB.
-

QUESTION 02:
Before the researchers give you real images to work with, you would like to test out any code you write. To this end, you would like to create "fake" simulated images and pretend they were captured by the microscope and the dye sensor. Using the data structures you chose in (1) above, write code to create such simulated images. Try and be as realistic in the generated images as possible.
ANSWER 02: Implemented.

QUESTION 03:
Using the simulated images generated by the code you wrote for (2) above as input, write a function to compute whether a parasite has cancer or not.
ANSWER 03: Implemented.

QUESTION 04:
You give your code from (3) to the researchers, who run it and find that it is running too slowly for their liking. What can you do to improve the execution speed? Write the code to implement the fastest possible version you can think of for the function in (3).
ANSWER 04: Implemented.
Vectorization with NumPy:
Vectorization is the process of performing operations on entire arrays rather than their individual elements. This technique is best of numerical and scientific computing in Python, primarily used by NumPy, that provides efficient, array-based computation.
Image data can be naturally represented as arrays (2D for grayscale images, 3D for color images), making NumPy an excellent tool for image processing tasks. By applying operations to whole arrays at once, you can leverage NumPy's optimized C backend, significantly speeding up computations compared to Python loops over individual pixels or array elements.

To Implement Vectorization:

- Use NumPy Operations: Replace loops with NumPy array operations. For example, to calculate the sum of two images, simply use image1 + image2 instead of manually iterating over each pixel.
- Avoid Python Loops for Element-wise Operations: For operations that apply to each pixel, use NumPy's element-wise operations instead of Python loops.
- Pre-allocate Arrays: When possible, allocate arrays at their full required size in advance of populating them, rather than dynamically resizing them during processing.

Parallel Processing with Joblib or Concurrent.Futures:

Parallel processing involves dividing a task into subtasks that can be executed simultaneously across multiple CPU cores or even different machines. This approach can significantly reduce execution time for tasks that are not inherently sequential.
Many image processing tasks, especially in batch processing scenarios like analyzing a large set of images, can be executed independently. This independence makes these tasks ideal candidates for parallel processing, allowing for significant reductions in overall processing time.

To Implement Parallel Processing:

- Identify Independent Tasks: Determine which parts of your workflow can be executed in parallel. In image processing, this often means identifying operations that can be applied to different images or different parts of an image independently.
- Choose the Right Tool: concurrent.futures provides a high-level interface for asynchronously executing callables. The ThreadPoolExecutor and ProcessPoolExecutor classes are suitable for different types of tasks. For CPU-bound tasks, ProcessPoolExecutor is often more effective due to Python's Global Interpreter Lock (GIL). For I/O-bound tasks, ThreadPoolExecutor can be beneficial.
- Joblib: This library offers a simple way to parallelize loops, with a focus on CPU-bound tasks. It's particularly user-friendly for those already working within the scientific Python ecosystem.

Algorithm Optimization:
Algorithm optimization involves analyzing and improving existing algorithms for efficiency. This can mean reducing the computational complexity of the algorithm, removing unnecessary operations, or employing more suitable algorithms for the task at hand.
Efficient algorithms are crucial for processing high-resolution images, where the sheer amount of data can make even seemingly simple operations computationally expensive. Optimizing algorithms can lead to reductions in both processing time and resource consumption.

To implement Algorithm Optimization:

- Profiling: Use profiling tools to identify bottlenecks in your code. This step is crucial for understanding where optimizations will have the most significant impact.
- Simplify Calculations: Look for ways to reduce the complexity of calculations, such as by removing redundant operations or simplifying mathematical expressions.

  .

So by combining these strategies, we can significantly improve the execution speed of image processing and analysis tasks, making it possible to handle larger datasets more efficiently and effectively.

QUESTION 05:
What other compression techniques can you suggest for both types of images (parasite and dye)? How would they impact runtime? Can you compute actual runtime and storage costs for typical images (not oversimplified image such as a circle for the parasite, or simple straight lines or random points for dye) in your code? The measurements should be done on your computer with an actual image size of 100,000x100,000 pixels (and not a scaled down version).
ANSWER 05:
For handling and storing high-resolution images like those in medical imaging or microscopy, efficient compression techniques are essential not only to save storage space but also to optimize data transfer and processing times. Beyond Run-Length Encoding (RLE) and Sparse Matrix representations, we can use the following:

1. Wavelet Compression
Wavelet compression, used in JPEG 2000, involves transforming the image into the wavelet domain, then quantizing and encoding the wavelet coefficients. It's particularly effective for images where spatial frequency characteristics change across the image, as it can achieve higher compression ratios with less perceptible loss compared to traditional Fourier transform-based methods.

Compression and decompression times can be longer than simpler methods due to the complexity of the wavelet transform and the subsequent processing steps. However, the resulting compression ratios can significantly reduce file sizes and, consequently, the time required for storage and transmission.

It is suitable for both parasite and dye images, especially if the images have varying levels of detail or if retaining high-quality detail in certain regions is more important.

2. Lossless JPEG
Unlike standard JPEG, which is lossy, Lossless JPEG uses a predictive coding model where each pixel is predicted from its neighboring pixels, and only the difference from this prediction is stored.

The encoding process can be computationally intensive, leading to longer compression times compared to more straightforward methods. However, it provides a good balance between compression efficiency and image quality.

This method is well-suited for both types of images when it's crucial to preserve all original data without any loss.

3. Dictionary Compression (e.g., LZ77, LZ78, LZW)
These algorithms create a dictionary of data sequences found in the input data. As the data is processed, sequences that are already in the dictionary are replaced with shorter references to the corresponding dictionary entries.

Compression speed is generally fast, and decompression speed is very fast, making these algorithms suitable for scenarios where fast decoding is necessary.

Effective for images that contain repetitive patterns or structures, which is often the case in scientific images, including both parasite and dye images.

4. Deep Learning-based Compression
Utilizes neural networks to learn efficient representations of image data. Autoencoders, for instance, can be trained to compress images into a compact latent space representation and then decompress them back into the original space.

Training the model can be very time-consuming and requires significant computational resources. However, once trained, encoding and decoding images can be relatively fast, especially with GPU acceleration.

Potentially very effective for a specialized set of images (like parasite and dye images) if sufficient training data is available to train the model effectively.

COMPUTING RUNTIME AND STORAGE COSTS

Understanding the runtime and storage costs associated with various image compression techniques is crucial for selecting the best method for a particular application, especially when dealing with very high-resolution images like those in medical imaging or remote sensing.

Why It's Challenging to Compute These Costs Directly Here

- Computational Resources: High-resolution images, especially those with dimensions like 100,000 x 100,000 pixels, require significant computational power and memory for processing. The compression and decompression tasks for images of this size can exceed the capabilities of standard computing environments, particularly in terms of RAM and CPU/GPU processing power.
- GPU Acceleration: Many advanced compression algorithms, especially deep learning-based ones, benefit significantly from GPU acceleration. The absence of GPU resources in certain environments can make it infeasible to perform these tasks within a reasonable timeframe.
- Software and Libraries: Some compression techniques may rely on specific libraries or software that are not available in all computing environments. This can limit the ability to directly implement and test all algorithms in a uniform manner.

Benchmarking Compression Methods in Real-World Applications

- To effectively evaluate the runtime and storage costs of different compression methods for high-resolution images, a structured benchmarking process is necessary:
- Implementation: The first step is to either implement the compression algorithms from scratch or utilize existing implementations. For many standard techniques, libraries and tools are available that provide optimized implementations. For custom or novel methods, especially those based on deep learning, implementation from scratch using frameworks like TensorFlow or PyTorch might be required.
- Testing Dataset: Selecting a representative set of images is crucial for benchmarking. The dataset should reflect the characteristics of images typical to the application, including

resolution, image content, and variability. For very high-resolution images, it might be necessary to work with subsets or downsampled versions during the development phase to make the process manageable.

- Measurement: The benchmarking process involves running the compression and decompression algorithms on the test dataset and recording key metrics:
    - Compression Time: The time it takes to compress the images.
    - Decompression Time: The time it takes to decompress the images back to their original form.
    - Compression Ratio: The size of the compressed image file relative to the original. This is crucial for understanding storage savings.
    - Quality Metrics: For lossy compression techniques, measuring the quality of the decompressed image compared to the original using metrics like PSNR (Peak Signal-to-Noise Ratio) or SSIM (Structural Similarity Index) is important.
- Analysis: Compare the performance of different compression methods across these metrics. The optimal method for a given application depends on the specific requirements for compression speed, decompression speed, storage savings, and image quality.

Considerations

- Trade-offs: There is often a trade-off between compression ratio, image quality, and the computational complexity of the algorithm. Higher compression ratios might come at the cost of reduced image quality or increased processing time.
- Application Requirements: The choice of compression method should be guided by the application's specific needs. For instance, real-time applications may prioritize fast decompression speeds, while archival projects might focus on maximizing storage efficiency.
- Ultimately, selecting the right image compression technique requires a careful evaluation of these factors, balancing the need for efficiency with the requirements for image quality and processing speed.

So ultimately, selecting the right image compression technique would require a careful evaluation of these factors, balancing the need for efficiency with the requirements for image quality and processing speed.

QUESTION 06:

Describes what tools you used to solve the challenge, particularly any LLM techniques.

ANSWER 06:

Online Resources: Google, GeeksforGeeks, Stack Overflow.

LLMs: Claude AI and Perplexity.

Logical Reasoning and Problem-Solving: Breaking down the problem and identifying appropriate approaches.

Just wanted to inform Claude AI and Perplexity were instrumental in solving this challenge.