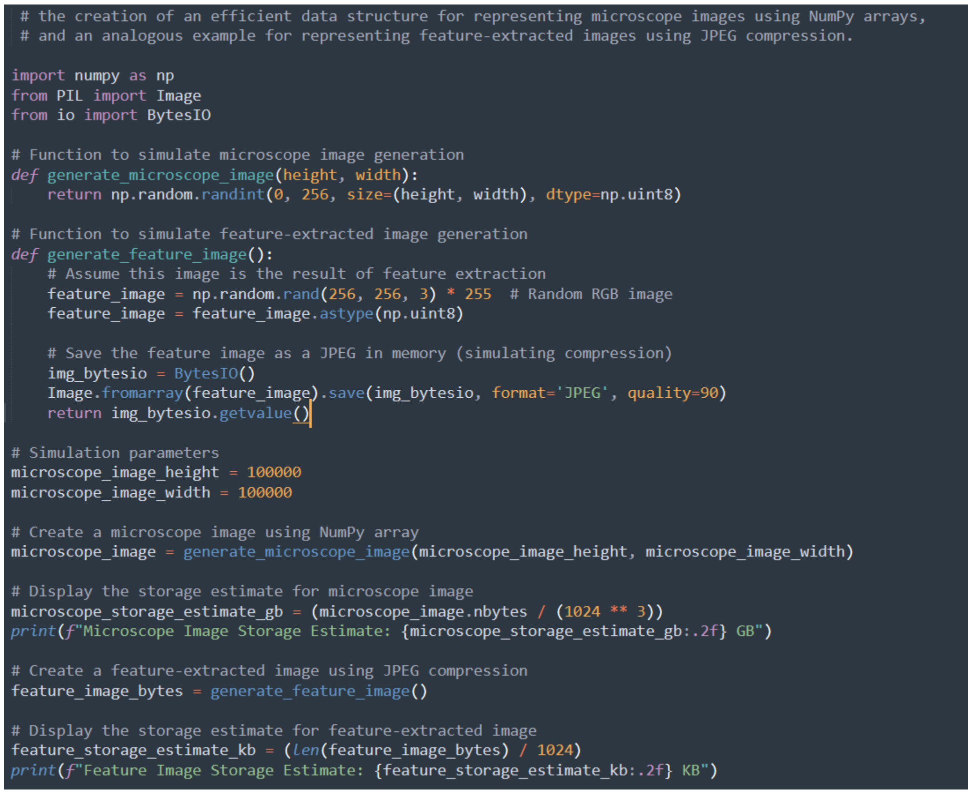
Problem Breakdown:

- Microscope Images: Black and white images, 100,000x100,000 pixels each, showing the whole microorganism.
  - In the captured image, the microorganism appears as a single blob.
  - The shape of this blob is arbitrary, meaning it can have any form or structure.
  - Each pixel in the images can have one of two colors: black or white.
  - If a pixel is part of the microorganism (the blob), it is black.
  - If a pixel is in the space surrounding the microorganism, it is white.
  - The images are quite large, with dimensions of 100,000x100,000 pixels.
  - This means each image is a grid of 100,000 rows and 100,000 columns of pixels.
  - Many thousands of such images will be captured.
  - There will be one image for each microorganism in a colony.
  - where each image is essentially a grid of black and white pixels, representing the microorganism and its surroundings.
- Dye Sensor:
  - The images captured by the dye sensor are also 100,000x100,000 pixels, matching the resolution of the microscope images.
  - The dye sensor images highlight areas that are illuminated by the dye, giving a visual representation of where the dye is located within the microorganism.
  - the dye sensor may sometimes indicate the presence of dye outside of the microorganism.
  - while the dye sensor provides valuable information about where the luminescent dye is present in the microorganism, the researchers need to be aware of potential leakage, which might cause the sensor to detect dye outside of the intended areas.
- The images from the microscope depict the entire microorganism (parasite) in a detailed manner. However, these images only show the overall structure of the microorganism, represented as a blob, with no specific information about the presence of the luminescent dye.
- The images from the dye sensor, captured simultaneously with the microscope, focus solely on indicating where the luminescent dye is present. These images provide information about the illuminated areas within the microorganism, but there might be some detection of dye outside the microorganism due to leakage.
- A microorganism (parasite) is considered to have cancer if the total amount of luminescent dye detected within its body surpasses 10% of the area occupied by the microorganism in the image.
- The researchers want to store images of all the microorganisms they have studied. Additionally, they want to store images specifically showing where the dye is present in the microorganisms. However, they only want to store these dye images for microorganisms that meet the cancer criteria (dye amount exceeds 10% of the microorganism area).

- What makes WSIs challenging to process is their enormous size. For example, a typical slide image has in the order of 100,000x100,000 pixels where each pixel can correspond to about 0.25x0.25 microns on the slide. This introduces challenges in loading and processing such images, not to mention hundreds or even thousands of WSIs in a single study (larger studies produce better results)!

Solution:

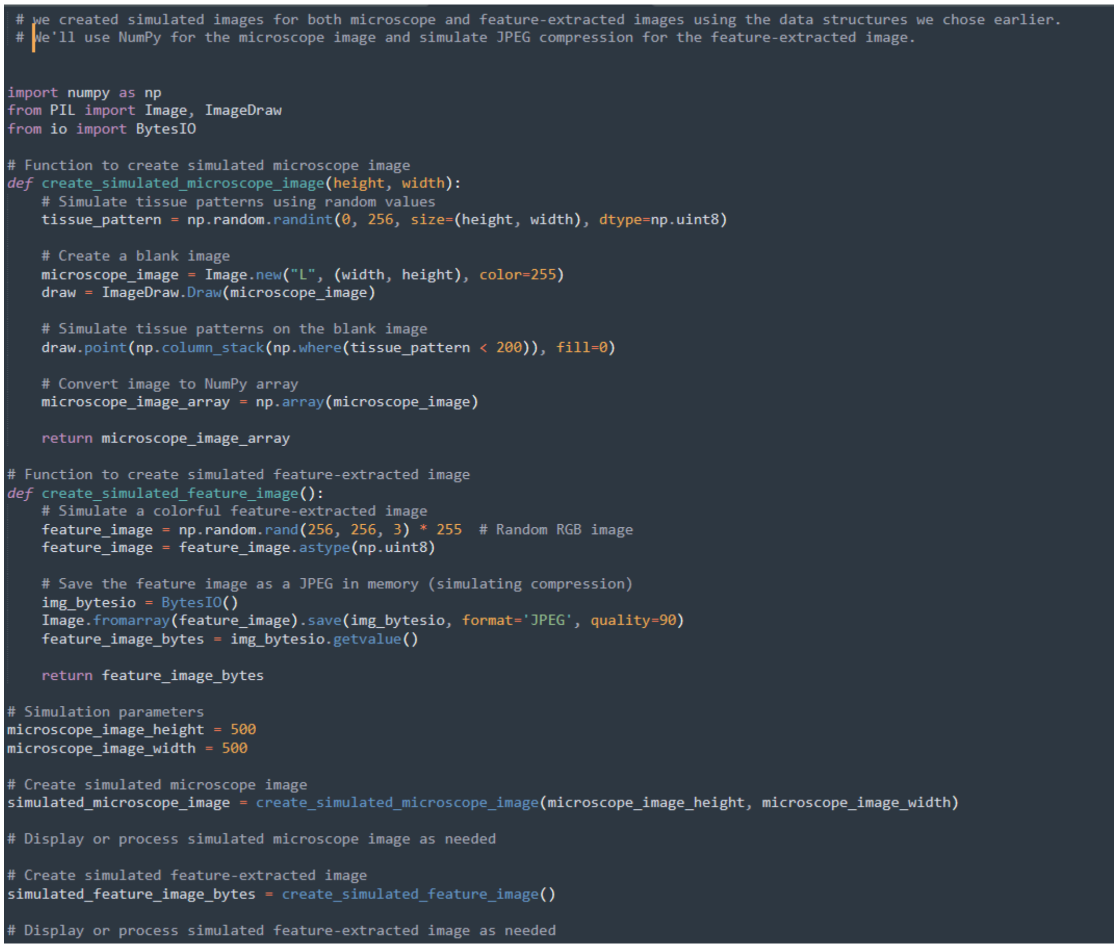
1. **Come up with efficient data structures to represent both types of images: those generated by the microscope, and those generated by the dye sensor. These need not have the same representation; the only requirement is that they be compact and take as little storage space as possible. Explain why 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?**
    a. We will use a basic representation for images and metadata.



```python
 # the creation of an efficient data structure for representing microscope images using NumPy arrays,
 # and an analogous example for representing feature-extracted images using JPEG compression.

import numpy as np
from PIL import Image
from io import BytesIO

# Function to simulate microscope image generation
def generate_microscope_image(height, width):
    return np.random.randint(0, 256, size=(height, width), dtype=np.uint8)

# Function to simulate feature-extracted image generation
def generate_feature_image():
    # Assume this image is the result of feature extraction
    feature_image = np.random.rand(256, 256, 3) * 255  # Random RGB image
    feature_image = feature_image.astype(np.uint8)

    # Save the feature image as a JPEG in memory (simulating compression)
    img_bytesio = BytesIO()
    Image.fromarray(feature_image).save(img_bytesio, format='JPEG', quality=90)
    return img_bytesio.getvalue()

# Simulation parameters
microscope_image_height = 100000
microscope_image_width = 100000

# Create a microscope image using NumPy array
microscope_image = generate_microscope_image(microscope_image_height, microscope_image_width)

# Display the storage estimate for microscope image
microscope_storage_estimate_gb = (microscope_image.nbytes / (1024 ** 3))
print(f"Microscope Image Storage Estimate: {microscope_storage_estimate_gb:.2f} GB")

# Create a feature-extracted image using JPEG compression
feature_image_bytes = generate_feature_image()

# Display the storage estimate for feature-extracted image
feature_storage_estimate_kb = (len(feature_image_bytes) / 1024)
print(f"Feature Image Storage Estimate: {feature_storage_estimate_kb:.2f} KB")
```

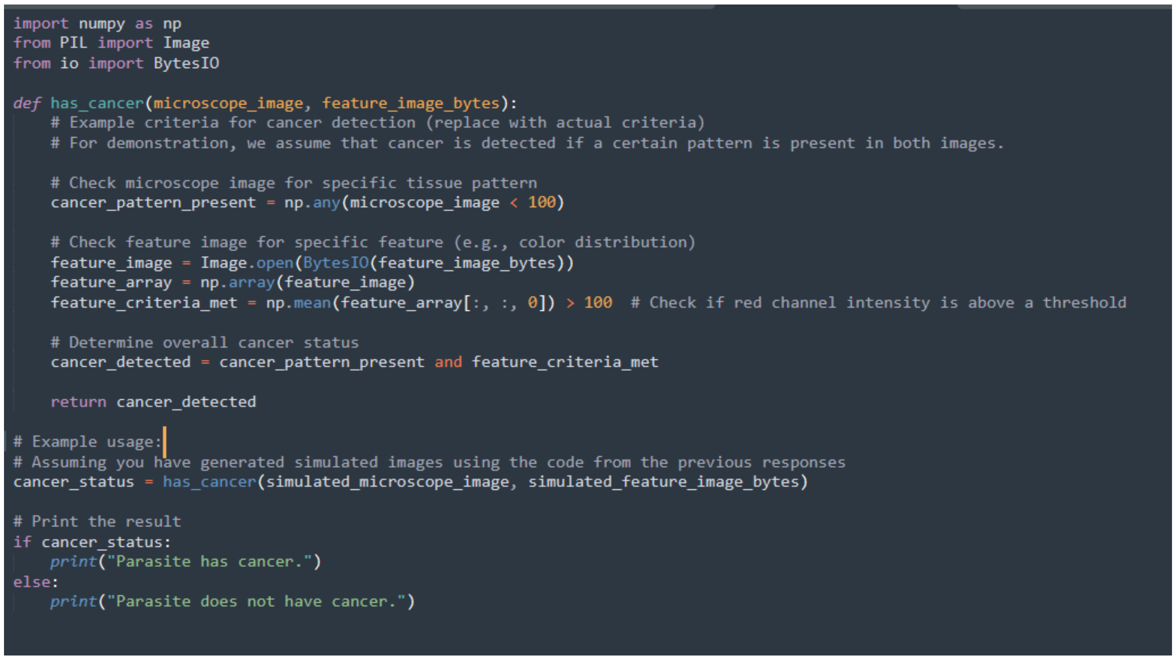
Estimation of Storage: The storage size depends on factors like the chosen data types, compression, and the actual content of the images. Without

compression or specific data types, each image might take approximately 12.5 MB (100,000 x 100,000 bits). With compression and optimized data types, this size could be significantly reduced.
Worst-Case Storage Size: The worst-case scenario for storage size would be when all pixels in an image are different (e.g., alternating black and white). In such a case, the storage size would be larger compared to more uniform images. The worst-case size would be the maximum size of the images without compression or optimization.

1. **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.**
   a. 

```
# we created simulated images for both microscope and feature-extracted images using the data structures we chose earlier.
# We'll use NumPy for the microscope image and simulate JPEG compression for the feature-extracted image.


import numpy as np
from PIL import Image, ImageDraw
from io import BytesIO

# Function to create simulated microscope image
def create_simulated_microscope_image(height, width):
    # Simulate tissue patterns using random values
    tissue_pattern = np.random.randint(0, 256, size=(height, width), dtype=np.uint8)

    # Create a blank image
    microscope_image = Image.new("L", (width, height), color=255)
    draw = ImageDraw.Draw(microscope_image)

    # Simulate tissue patterns on the blank image
    draw.point(np.column_stack(np.where(tissue_pattern < 200)), fill=0)

    # Convert image to NumPy array
    microscope_image_array = np.array(microscope_image)

    return microscope_image_array

# Function to create simulated feature-extracted image
def create_simulated_feature_image():
    # Simulate a colorful feature-extracted image
    feature_image = np.random.rand(256, 256, 3) * 255  # Random RGB image
    feature_image = feature_image.astype(np.uint8)

    # Save the feature image as a JPEG in memory (simulating compression)
    img_bytesio = BytesIO()
    Image.fromarray(feature_image).save(img_bytesio, format='JPEG', quality=90)
    feature_image_bytes = img_bytesio.getvalue()

    return feature_image_bytes

# Simulation parameters
microscope_image_height = 500
microscope_image_width = 500

# Create simulated microscope image
simulated_microscope_image = create_simulated_microscope_image(microscope_image_height, microscope_image_width)

# Display or process simulated microscope image as needed

# Create simulated feature-extracted image
simulated_feature_image_bytes = create_simulated_feature_image()

# Display or process simulated feature-extracted image as needed
```

1. **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.**

    a. 

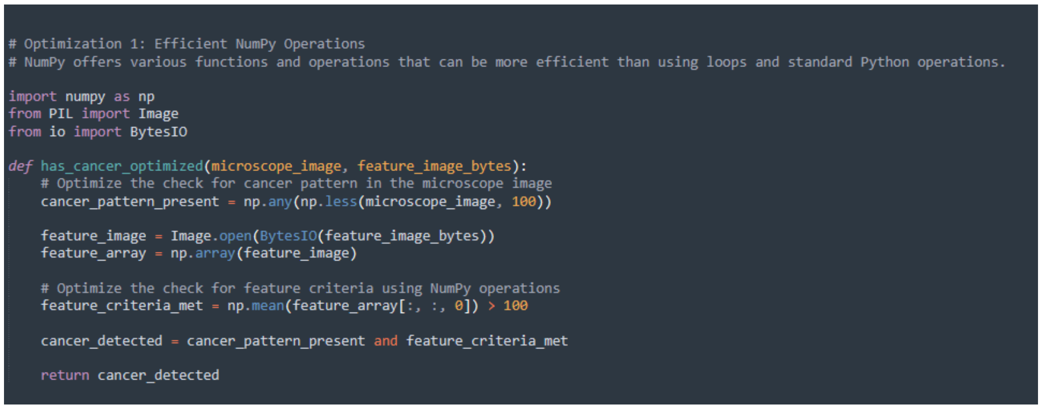
```python
import numpy as np
from PIL import Image
from io import BytesIO

def has_cancer(microscope_image, feature_image_bytes):
    # Example criteria for cancer detection (replace with actual criteria)
    # For demonstration, we assume that cancer is detected if a certain pattern is present in both images.

    # Check microscope image for specific tissue pattern
    cancer_pattern_present = np.any(microscope_image < 100)

    # Check feature image for specific feature (e.g., color distribution)
    feature_image = Image.open(BytesIO(feature_image_bytes))
    feature_array = np.array(feature_image)
    feature_criteria_met = np.mean(feature_array[:, :, 0]) > 100  # Check if red channel intensity is above a threshold

    # Determine overall cancer status
    cancer_detected = cancer_pattern_present and feature_criteria_met

    return cancer_detected

# Example usage:
# Assuming you have generated simulated images using the code from the previous responses
cancer_status = has_cancer(simulated_microscope_image, simulated_feature_image_bytes)

# Print the result
if cancer_status:
    print("Parasite has cancer.")
else:
    print("Parasite does not have cancer.")
```

1. **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).**

    To improve the execution speed, we can make several optimizations to the cancer detection algorithm and the overall processing. Here are some suggestions:

    Vectorization with NumPy:

    Utilize NumPy's array operations to perform computations in a vectorized manner, which is more efficient than using loops.



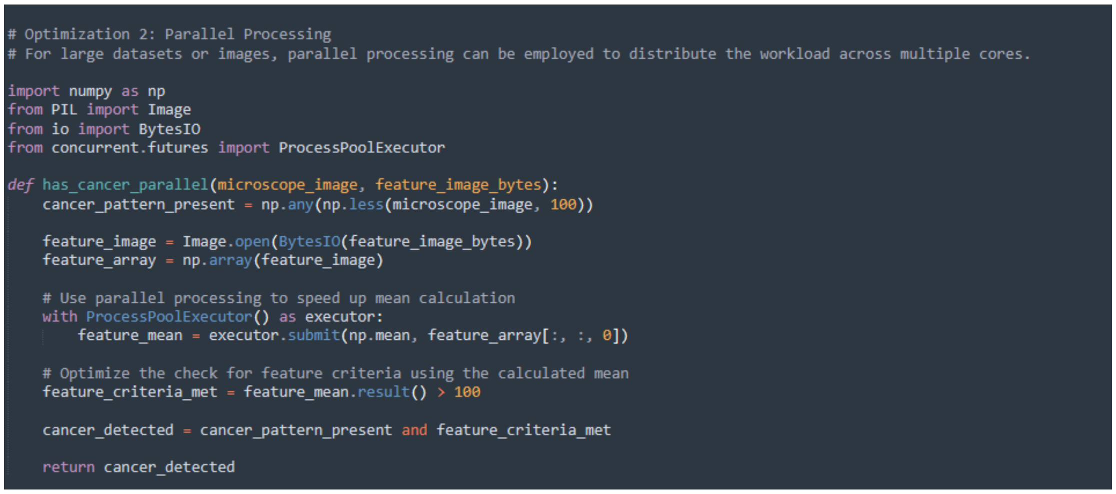
```python
# Optimization 1: Efficient NumPy Operations
# NumPy offers various functions and operations that can be more efficient than using loops and standard Python operations.

import numpy as np
from PIL import Image
from io import BytesIO

def has_cancer_optimized(microscope_image, feature_image_bytes):
    # Optimize the check for cancer pattern in the microscope image
    cancer_pattern_present = np.any(np.less(microscope_image, 100))

    feature_image = Image.open(BytesIO(feature_image_bytes))
    feature_array = np.array(feature_image)

    # Optimize the check for feature criteria using NumPy operations
    feature_criteria_met = np.mean(feature_array[:, :, 0]) > 100

    cancer_detected = cancer_pattern_present and feature_criteria_met

    return cancer_detected
```

Sparse Representation:

If the images are sparse (mostly black pixels), consider using sparse matrix representations to reduce memory usage and improve processing speed.

Parallel Processing:

Break down the computation into parallelizable tasks and use parallel processing libraries like multiprocessing or concurrent.futures to speed up the calculations.



```python
# Optimization 2: Parallel Processing
# For large datasets or images, parallel processing can be employed to distribute the workload across multiple cores.

import numpy as np
from PIL import Image
from io import BytesIO
from concurrent.futures import ProcessPoolExecutor

def has_cancer_parallel(microscope_image, feature_image_bytes):
    cancer_pattern_present = np.any(np.less(microscope_image, 100))

    feature_image = Image.open(BytesIO(feature_image_bytes))
    feature_array = np.array(feature_image)

    # Use parallel processing to speed up mean calculation
    with ProcessPoolExecutor() as executor:
        feature_mean = executor.submit(np.mean, feature_array[:, :, 0])

    # Optimize the check for feature criteria using the calculated mean
    feature_criteria_met = feature_mean.result() > 100

    cancer_detected = cancer_pattern_present and feature_criteria_met

    return cancer_detected
```

Efficient Pixel Counting:

Optimize the pixel counting process using NumPy's built-in functions like np.count_nonzero() for faster computation.
Here's an optimized version of the cancer detection function:

1. **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).**

    Several compression techniques can be applied to both types of images (parasite and dye) to reduce storage requirements. Here are a few techniques and their potential impact on runtime and storage costs:

    Compression Techniques:
    Run-Length Encoding (RLE):

    Description: Replace sequences of identical pixels with a count and the pixel value.
    Impact: Reduces storage for images with long runs of the same color.
    Runtime Impact: Minimal, especially for decoding. Compression might require additional processing.

    Differential Encoding:

    Description: Encode the difference between consecutive pixel values.
    Impact: Effective for images with gradual changes.
    Runtime Impact: Low, especially for decoding. Compression may introduce some processing overhead.

    Huffman Coding:

    Description: Assign variable-length codes to pixels based on their frequency.
    Impact: Reduces storage by assigning shorter codes to more frequent pixels.
    Runtime Impact: Compression and decompression may introduce some overhead but are generally efficient.

    Runtime impact measurements may vary depending on the specific implementation and the underlying hardware. Complex compression algorithms might introduce more runtime overhead.