**Architecture Document**

**Image Data Processing Solution**

Image data access

* **An IImage** interface is used for accessing image data, ensuring flexibility and extensibility. Clients are required to provide a specific implementation, which allows querying data from external sources and supports various data formats. This design accommodates both big-endian and little-endian systems.
* Two implementations of the **IImage interface** are provided: **VectorImage** for testing purposes, which simulates image data in a vector, and another **ImageWrapper** for real-time, in-memory data manipulation for live scenarios.

Image data processing

* To maintain high performance, the solution processes image data directly in memory without copying it. This approach necessitates that the image data remain constant during processing to avoid incorrect outcomes. If the data is likely to change, it must be copied before processing, if not wrong results outcomes
* A processing interface is defined, with a specific **GoProImageProcessor** logic implementation designed to identify the top 50 pixel values in an image

Special Considerations and Assumptions

1. **Positive Value Range:** The system assumes that all pixel values are positive.
2. **Sorting Not Required**: The solution does not sort results by pixel values, as this is not explicitly required by the specifications.
3. **Top Distinct vs. Top N:** The problem statement does not specify the need for the top 50 *distinct* values. Thus, it is assumed acceptable to include pixels with the same color value but in different positions within the top N results.
4. **Handling Identical Values:** When multiple pixels share the same value and rank within the top 50, the solution can yield multiple correct outcomes. In such cases, any pixel that falls within the top 50 and has an equivalent value outside the top selection is considered a valid candidate. The current solution prioritizes these pixels based on their order of appearance to simplify logic and ensure deterministic results, particularly beneficial for unit testing and handling values.

**Ex:** If the image has fewer than 50 distinct pixel values but more than 50 pixels overall, the solution will return pixels with identical values in different positions to fill the top 50 slots. This means the top 50-pixel values may include duplicate values but from different locations.

1. **Images with Size Less Than Top N**

For an image with uniform pixel values (all zero), it is acceptable to return any 50 pixels. If the image contains fewer than 50 pixels, all pixels are returned, even if they do not constitute the top N values because the total number of pixels and consequently the number of outcomes is less than N.

Performance Considerations

**Multiple Queries on the Same Image:** If the system performs multiple queries for the top N values on the same image, it may benefit from sorting the image by pixel values before querying. This pre-sorting allows for efficient repeated top N value queries.

**Single Query Across Multiple Images:** The current implementation performs in scenarios where a single query is made across many new images. The algorithm considers a direct heap data structure, which performs well on average, even in special cases, regardless of the top N size relative to the image itself.

**Top N vs. Image Size**

If **small** **top** **N** (**topN / total\_pixels < 0.1)**, consider partial sort, or considering special cases were calculating the maximum values for top N steps (excluding previous maxima) can yield better performance than the general approach.

For **big top N (topN / total\_pixels > 0.5)**, where execution speed is critical and size of color values are acceptable, for example - *16-bit word equating to a pixel value -* , consider using a counting sort algorithm with a complexity of O(n) for sorting.

**Single Core vs. Multicore Environments:** In multicore environments, consider dividing the image into chunks and assigning each chunk to a separate core for processing. The results from each core should then be combined (e.g., using merge sort) into a final, merged output, optimizing performance across multiple processing unit

**BIG Image size**: Consider a distributed processing strategy by dividing the image into several smaller segments. Then execute processing with segments then join results.

Optimization – Constant values make differences!

to achieve the best performance in terms of speed and memory I implemented 3 types of algorithms, using heap, using set and using counting sort. All of them was fine tuning.

Analysis in terms of theoretical math and algorithm complexities strategies say no more difference but o real scenarios at production ready with concrete implementation my orchestration show a real challenge to choose relevant key points for improvements.

Then changes was to define a math formula – polynomial – to decide on specific scenarios which is the best choosing algorithm.

Based on other research and publications [Applied Sciences | Free Full-Text | A General Framework Based on Machine Learning for Algorithm Selection in Constraint Satisfaction Problems (mdpi.com)](https://www.mdpi.com/2076-3417/11/6/2749)

I made a prof of concept based on IA with random forest.

Stept 1:

generate relevant input data by using that image matrix :

Random Matrix, "Sorted Ascending", "Sorted Descending,"With Specific Distribution","Uniform Distribution"

Step 2

Bencmarking and Collecting data output data – exeution time per strategy selector.

Step 3

Training model

Step 4

Model

Step 5 using model in c++ on real scenarios.