3. WRITTEN RESPONSES

3 a.

3.a.i.

The project's name is AIMER which stands for 'Artificial Intelligence Mark Evaluator & Recognizer'. In short, its purpose is: a
ssisting teachers/educators in grading their students' Multiple Choice Question ("MCQ") answer papers through a simplified and automated manner, rather than having to review them manually or the traditional way.

3.a.ii.

All that there is to be encountered/viewed by the user during their usage of this application, including: double-clicking the application for running, holding the camera at a suitable height from the student's paper, and lastly viewing the answer key side-by-side the finalized grade.

3.a.iii.

Input: Student-produced MCQ answer sheet; answer key provided by the teacher.
Output: Finalized grade (percentage 0-100%) of the student based on the two aforementioned inputs.

In our sample case, the answer key contained:
First=1(A)
Second=3(C)
Third=4(D)
Fourth=2(B)
Fifth=1(A)

And the student-produced answer sheet contained:
First=1(A)
Second=3(C)
Third=2(B)
Fourth=2(B)
Fifth=4(D)

3 b.

3.b.i.

```python
def extract_answers_from_paper(paper):
    paper = cv.cvtColor(paper, cv.COLOR_BGR2GRAY)
    paper = cv.threshold(paper, 127, 255, cv.THRESH_BINARY_INV)[1]


    # The answer grid is a 2-D array filled with zeros at first that will later hold the 'weight' of choices.
    answer_grid = np.zeros( (NUM_QUESTIONS, NUM_CHOICES) )
    # Evenly splits the paper into seperate rows of questions, each having a number of choices.
    questions = list(np.array_split(paper, NUM_QUESTIONS, axis=0)) #np.vsplit(paper, NUM_QUESTIONS)


    for count_question, question in enumerate(questions):
        # Evenly splits every column of choices in each row into multiple 'cells' or choice.
        choices = list(np.array_split(question, NUM_CHOICES, axis=1)) #np.hsplit(question, NUM_CHOICES)
        for count_choice, choice in enumerate(choices):
            # Stores the 'weight' of the choice/cell (i.e., how many filled pixels it had.)
            answer_grid[count_question][count_choice] = cv.countNonZero(choice)

    # Since each cell is more or less going to have a number of filled-in pixels, we will
    # find the maximumly-filled area along the x-axis or the choices of each question.
    answer_list = np.argmax(answer_grid, axis=1)

    #cv.namedWindow("test")
    #while True:
    #    cv.imshow("test", question)
    #    if (cv.waitKey(33) == ord('z')):
    #        break

    return answer_list

```

3.b.ii.

```python
    for count_question, question in enumerate(questions):
        # Evenly splits every column of choices in each row into multiple 'cells' or choice.
        choices = list(np.array_split(question, NUM_CHOICES, axis=1)) #np.hsplit(question, NUM_CHOICES)
        for count_choice, choice in enumerate(choices):
            # Stores the 'weight' of the choice/cell (i.e., how many filled pixels it had.)
            answer_grid[count_question][count_choice] = cv.countNonZero(choice)

```

3.b.iii.

questions

3.b.iv.

Said list is a 2-D array. In other word, the scanned image of a paper is divided into separate rows (list) of questions, each then having a number of choices.

3.b.v.

Without having it, the user would be forced to manually move the camera over each and every answer bubble of the student's answer sheet, this is hugely counterintuitive, inconvenient, and defeats the entire automation and time-saving goal of the program.

3 c.

3.c.i.

```python
# 480p for a higher processing speed
WIDTH = 640
HEIGHT = 480
# The margin or outline/border of the document
MARGIN = 25 # TODO: Obtain this automatically from the image dimensions and ratios.

def scan(img):
    img = cv.resize(img, (WIDTH, HEIGHT))
    img_original = img

    # TODO: lower the brightness and preserve edges.
    #img = cv.bilateralFilter(img, 11, 17, 17)
    img = cv.Canny(img, 200, 200)

    kernel = np.ones((5, 5))
    img = cv.dilate(img, kernel, iterations=2)
    img = cv.erode(img, kernel, iterations=1)

    #cv.namedWindow("debug")
    #cv.imshow("debug", img)


    # Extracts all the shapes ("contours") from the image.
    contours, _hierarchy = cv.findContours(img, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)

    largest_countors = find_largest_contours(contours) # Finds the largest contour.
    if largest_countors.size == 0: # No shapes were detected.
        return img_original, np.zeros((HEIGHT, WIDTH, 3), np.uint8) # So, we return a blank image.

    img_bordered = img_original
    cv.drawContours(img_bordered, [largest_countors.astype(int)], -1, (0, 255, 0), 2)

    largest_countors = sort_points(largest_countors) # Sorts its points.

    # De-skew the image
    src = np.float32(largest_countors) # Coordinates of the largest shape (i.e., the paper or document).
    dst = np.float32([[0, 0], [WIDTH, 0], [0, HEIGHT], [WIDTH, HEIGHT]]) # Same size as the resized img.
    matrix = cv.getPerspectiveTransform(src, dst)
    img_scanned = cv.warpPerspective(img_original, matrix, (WIDTH, HEIGHT))


    # Crops the blurry outline/margin.
    img_scanned = img_scanned[MARGIN:img_scanned.shape[0]-MARGIN, MARGIN:img_scanned.shape[1]-MARGIN]
    img_scanned = cv.resize(img_scanned, (WIDTH, HEIGHT))

    # Anti-aliasing
    img_scanned = cv.medianBlur(img_scanned, 9)

    return img_bordered, img_scanned
```

3.c.ii.

```python
def main():
    cv.namedWindow("webcam")
    cv.namedWindow("graded")

    img = None
    scanned = None

    #image = cv.imread("test.png", cv.IMREAD_GRAYSCALE)

    while True:
        _ret, img = cap.read()

        img, scanned_img = scanner.scan(img)
        cv.imshow("webscam", img)

        score, graded_img = grader.grade(scanned_img)
        cv.imshow("graded", graded_img)

        if (cv.waitKey(33) == ord('s')):
            break

    print("The final grade was: " + str(score) + "%")
```

3.c.iii.

This subroutine takes any image that may contain extra background objects than just an answer sheet paper, and then proceeds on heavily enhancing, de-noising, de-skewing, and cropping the paper from the overall image in order for it to be suitable for further image recognition and grading procedures.

3.c.iv.

(Important note: The below steps require the *OpenCV-Python* library to be installed.)

1. Resize the input image using cv.resize() to an optimal size depending on your available processing power (480p resolution is recommended), after doing so, prepare a copy of the initial image in a separate variable such as img_original. **(This avoids wasting computer processing power and does not reduce accuracy.)**
2. Apply an edge detector algorithm such as cv.Canny() and then, in order to get rid of the unwanted noise, dilate and erode the resulting outlines by using cv.dilate() and cv.erode(), respectively. **(This filters out the rough edges and artifacts of the image that might otherwise interfere with our scanning procedure in later stages of the program.)**
3. By using cv.findContours() on the previous step's output, get the coordinates of all shape shapes found by the canny algorithm. **(Keep in mind that this will return ALL of the shapes, we are still going to have to find out the paper edges from these in the next step.)**
4. By iterating through the list containing all shapes, find the largest one by surface area using cv.arcLength() and cv.approxPolyDP(), and keep its edge coordinates. **(The largest contour/shape is the desired paper's location. Also, you might want to safeguard against the case where no shapes/contours are detected by returning a 'sentinel' value such as all zeroes, this ensures that your program will not have undefined behaviour.)**
5. <u>Optionally,</u> use the cv.drawContours on img_original from step 1 to make the detected paper visible on the background image, you may later return this alongside the final scanned paper.
6. Using cv.getPerspectiveTransform(src, dst), where src and dst are the coordinates of the largest contour and the dimensions of a 480p matrix, crop and de-skew the paper into its own standalone image. **(Now we are almost done and ready.)**
7. <u>Optionally</u>, cut a few pixels of margin out of the image so that the remnants of the underlying background are not visible.
8. Apply FXAA anti-aliasing using cv.medianBlur(). **(So that the effects of downsampling will not be obvious to the human eye.**)
9. Lastly, return the cropped image of the paper. **(<u>Optionally</u>, return the image from step 5 with the paper bordered in a bright color)**

3 d.

3.d.i.

First call:

First Case: A proper image containing an actual paper is supplied to the scan() subroutine.

Second call:

Second Case: An improper image (that is, an image that has NO paper that might contain other objects such as a flower pot) is supplied.

3 d.ii.

Condition(s) tested by first call:

Usage of the algorithm in order to receive a scanner-quality image of the paper, given an image taken by a webcam.

Condition(s) tested by second call:

Whether or not the procedure is properly safeguarded and can protect the program against crashes or undefined behaviour from unexpected inputs.

3.d.iii.

Results of the first call:

scan() does all work as usual and returns the expected output.

Results of the second call:

Results in normal execution up until **step 4**, then, upon finding no papers, returning the original image (without any borders) alongside a **completely blank** matrix (with all zeroes), which indicates that no papers or rectangular shapes were found.