ASSIGNMENT 1 REPORT

How to run the program:

```
make
./omr <image_name>
```

For large images such as the given *rach.png*, the program may run for a couple of minutes.

The output of the program are:

- *scores4.png*: The result of the convolution in step 4.
- *detected4.png*: The visualization of the detected symbols after step 4.
- *edges.png*: The image edge map produced in step 5.
- *edges_thin.png*: Extra image after the non-maximum suppression has been applied to the edges.png.
- *detected5*.png: The visualization of the detected symbols after step 5.
- *staves.png*: The original image, with detected staves marked with blue lines.
- *detected7.png*: Visualization of which notes were detected that was discovered using the combination of step 4 and step 5.
- *detected7.txt*: A text indicating the detection results.

Rescaling

For the rescaling, we use the ratio of the interval between staff lines to the height of the note template. If the ratio is below 2 (the resolution of the music sheet image is 2 times to the resolution of the template), then we rescale the templates to match the music sheet. Otherwise, we rescale the music sheet. The reason is that by scaling up the template rather than scaling down the template, we can achieve a better accuracy. However, we don't want the template to become too large, otherwise it will take a long time to compute convolutions. In addition, here we assume the music sheet's resolution shouldn't be 5 times or higher that the template's.

One thing should be cared about here is the aliasing problem, that the scale ratio might be something that simple subsampling will cause problems, like 9/10. Simple subsampling will change the shape of the graph. Our solution is to do a 4 times sample according to the begin position and end position of the pixel on the original image, and then average the values. If the scale is under 1/2, the program averages all the pixels within corresponding region. This works well on the templates, but has a little worse performance on the image. So we hope we don't need to rescale the image.

Hamming distance
To calculate the Hamming distance, the program uses the continuous version of the Hamming distance function, instead of the binary version provided in the guidance. To do this we need first normalize the gray scale value to range [0,1] by dividing it by 255. Then the formula is the same.

Then, we need to figure out where the note is. The program do a non-maximum suppression on the Hamming distance map with a threshold and region parameters(width and height) and then find where there is a value on the map.

The non-maximum suppression first requires the Hamming distance at one position is larger than the threshold. Then it requires there doesn't exist another point is larger than this point. If there is a pixel having the same value, then it's required to not be above or on the left of this pixel. If all these requirement are satisfied, then this value is kept. Otherwise, it will be set to 0. After this there will be a spot on the map indicating that it should be recognized a note on the position.

The width of the region is set to the 0.5 times the width of the template, and the height is about 0.6 times the template's height. This is because some time the note will be recognized with a little offset, and we don't want they to be merged into one. The threshold is set to 0.75, 0.71, and 0.71 for note, quarter rest, and eighth rest separately.

Finding Pitch
To get the pitch of a note, first we need to find the first line of a staff. Then, decide which staff one note belongs. We assume one note should not be 4 times interval higher than the first line of its staff, and should be above such distance to the below staff. Then multiply the distance from one note to its staff's first line by 2 and divide by the interval between lines. According to this number and the base of the staff we can figure out the pitch of this note. Here we assuming that the staffs with odd number are based on G, and those with even number are based on B.

Hough Transform

Hough Transform is the method used to detect lines by allowing each pixel to vote for a line it is a part of.

The hough transform implementation used in our program is as follows:

- **Non Maximum Suppression to thin lines**

Before doing the hough transform each line of the staff is thinned to eliminate the noise around it.

This is done by making the window size for the maximum suppression to be of 0 width and 2 pixel high.

- **Finding votes and normalizing the votes using min-max normalization**

Each pixel in the image is allowed to vote for the y-coordinate it is a part of if the pixel has a value greater than the pixel_threshold assumed. So the number of y-coordinates is equal to the length of the image.

From the votes gathered we do a min max normalization using the formula:

x=(x-min)/(max-min)

- **Hough Voting Space**

After Finding the normalized votes, each pixel whose y-coordinate has a threshold greater than the normalization threshold is allowed to vote once for the line of the staff greater than normalization threshold before it and the space between the line and itself.

From this voting space we get the best space between the lines. We can also calculate the first line of the staff from this voting space.

Assumptions for hough transform

- The staff lines are assumed to be perfectly horizontal.
- For a given music score the space between all the staff lines are same.

When Hough Transform Performs the Best

Our Implementation of hough performs the best when all the staff lines are of the same size.

When Hough Transform is not efficient

Our implementation of hough is not as efficient when the staff lines are not of the same size.

How Hough Transform can be improved in the future

Hough Transform can be improved in the future by dynamically determining the normalization thresholds. This can be done by determining the outliers among the normalized votes by finding the lines which have normalization values which line after the 3rd quartile.

Edge detection and Template matching by edges:

We used sobel operator and separable convolution to get the edges of the image. The output of this step is provided in the: *edge.png* file.

After the edge detection, we used the edge template matching algorithm provided to find out the best matched templates. With plain regular edges we were having trouble finding good matches for the complex images. So we used a modified non-maximum suppression algorithm to thin the edges such that all edge pixels are 1 pixel width. We used a double pass non-maximum suppression where in the first pass, as a regular method we keep the pixels in the local maxima of the gradient angle. In the second pass, when two or more pixels are nearby with the same gradient angle, we merged them into one pixel, which results in a thin edge map (that we output in the *edges_thin.png*), and later we used the thin edge map to match the templates, which results in a better finding.

We used djikstra algorithm to find out the nearest edge pixel from any pixel of the main image, which is required for the edge based template matching algorithm.

We slightly modified the given algorithm for edge based template matching as well. In the original algorithm, when a template position has a pixel, it looks for in the same position in the main image, how far is an edge pixel, and does not do anything when the template position does not have a pixel. We modified it so that when a template position does not have pixel, it looks for the nearest pixel in the template, and nearest pixel in the main image, and compare how similar they are. This modified method gave us better result.

Performance of template matching by edges:

There are many cases when the notes are overlapping, a few notes are stacked on top of each other and overlapping. In this case, the edge detector detect only the left side and right side edges of the middle notes and failed to detect the top and bottom edges of the notes, as they are overlapped with other notes from top and bottom. This results in a poor performance by the edge based template matching method. Still we thin out the edges and carefully chose parameter to get as best result as possible.

When notes are not overlapping, the edge detector detect all four side edges of the notes and results in a good performance.

Joining the result of step 4 and step 5:

In step 4 and step 5, we gave every detected symbols a confidence value within 0 and 1. After we are done with the step 4 and step 5, we joined the result of them both together by their confidence values to get the final result of *detected7.png.* When both the result from step 4 and step 5 was positive for a symbol, we picked that symbol for the final result. We gave more weight to the step 4 result in the joining compared to step 5, as step 4 results were better most of the times than step 5. Also we generated the *detected7.txt* with all the detected symbol as required.

Final Results:

result for music1.png:

```
Mean average precision for filled_note = 1.000000
Mean average precision for eighth_rest = 1.000000
Mean average precision for quarter_rest = 0.000000  (Actually the recognition are correct, this is
due to some position offset. Please see the output image, detected7.png)
OVERALL Mean average precision = 0.666667
---
Fraction of correct note_head detections with correct pitch = 0.976190
```

result for music4.png:

```
Mean average precision for filled_note = 0.829139
Mean average precision for eighth_rest = 1.000000
OVERALL Mean average precision = 0.914569
---
Fraction of correct note_head detections with correct pitch = 0.766355
```