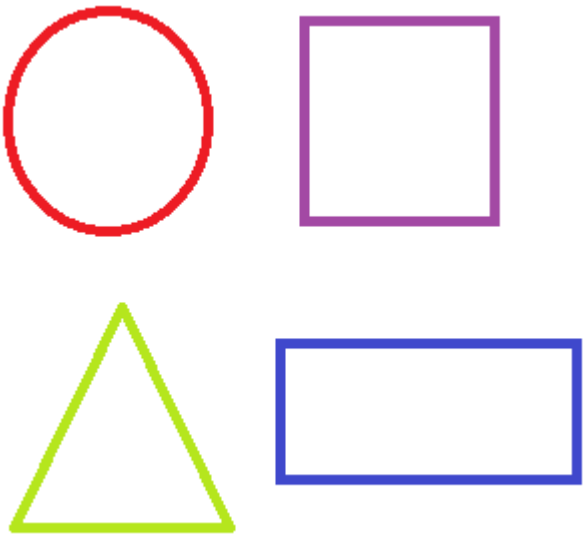
Beatriz Manrique CSE 3353

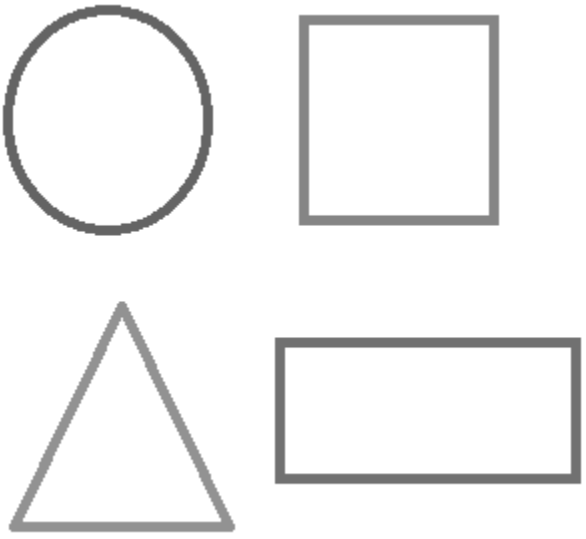
Project 3: Image Manipulation and Process Scheduling

Image Manipulation

Original Image (Figure 1)



Grayscale (Figure 2)



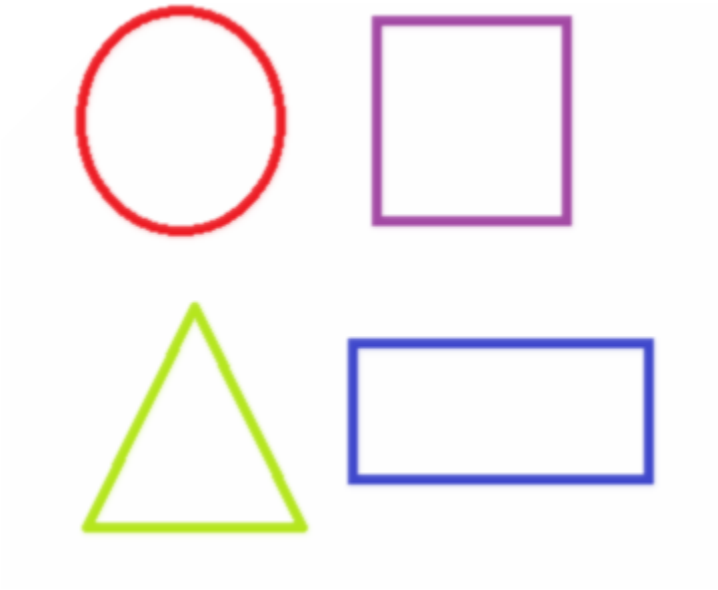
Converting an rgb image to a grayscale image requires an averaging calculation. Each pixel of an image has an individual Red, Green, and Blue value. Summing the Red, Green, and Blue values and then dividing by three averages the pixel

value to grayscale. The averaged value is then reassigned to each Red, Green, and Blue values. In turn, this makes the whole image grayscale.

To grayscale an image, each pixel must be visited. An image with *x* width and *y* length will have (*xy*) pixels, where *x* and *y* are constants. Therefore, the runtime of the grayscaling algorithm is O(*xy*). If we can find an *n* that is greater or equal to both *x* and *y*, the runtime can be written as O(n^2).

To execute the grayscale algorithm, enter “node grasycale.js”

Blur (Figure 3)



Blurring Kernel

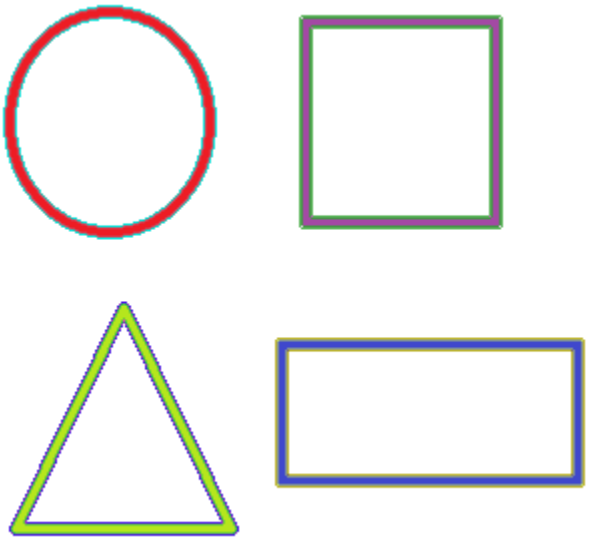
$$\begin{matrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{matrix}$$

Blurring an image is conceptually simple. Form a 3x3 matrix around the focus pixel. This means there will be a top left, top center, top, right, left, right, bottom left, bottom center, and bottom right pixel including the focus pixel as the center pixel to make the 3x3. Each pixel location corresponds to a value found on the blurring pixel shown above. According to their location, the rgb values of the pixel is multiplied by the kernel value and summed to a total. This method is applied to each pixel in the 3x3 matrix. Each pixel sum is then collectively summed to make a total sum. That total sum is then written to the Red, Green, and Blue values of the focus pixel. Once this method is applied to every pixel in the image, the result is a blurred copy.

To blur an image, each pixel must be visited. The multiplications and summations happen in constant time. An image with x width and y length will have (xy) pixels, where x and y are constants. Therefore, the runtime of the blurring algorithm is O(xy). If we can find an n that is greater or equal to both x and y, the runtime can be written as O(n^2).

To execute the blurring algorithm, enter “node blur.js”

Sharpen (Figure 4)



Sharpening Kernel

$$\begin{matrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{matrix}$$

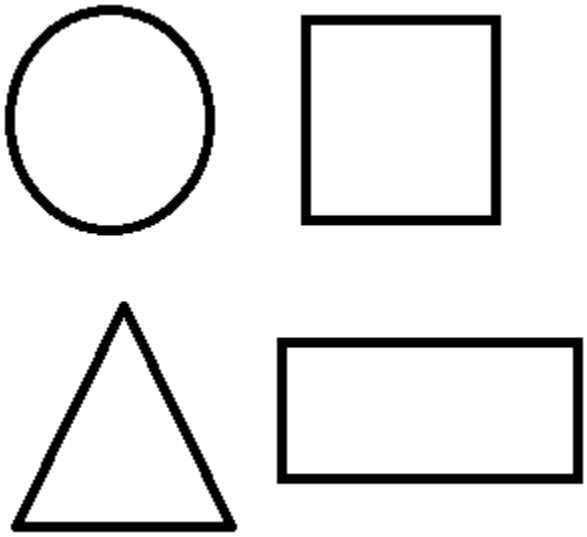
Sharpening an image is extremely similar to burring an image. The only difference is the kernel used. The sharpening kernel, shown above, and the 3x3 around the focus pixel are matrix multiplied much like the blurring method. A total sum is found and written to the Red, Green, and Blue values of the focus pixel. After applying to every pixel in the image, the result is a sharpened copy.

To sharpen an image, each pixel must be visited. The multiplications and summations happen in constant time. An image with x width and y length will have (xy) pixels, where x and y are constants. Therefore, the runtime of the sharpening algorithm is O(xy). If we can find an n that is greater or equal to both x and y, the runtime can be written as O(n^2).

To execute the sharpening algorithm, enter “node sharpen.js”

Connected Components

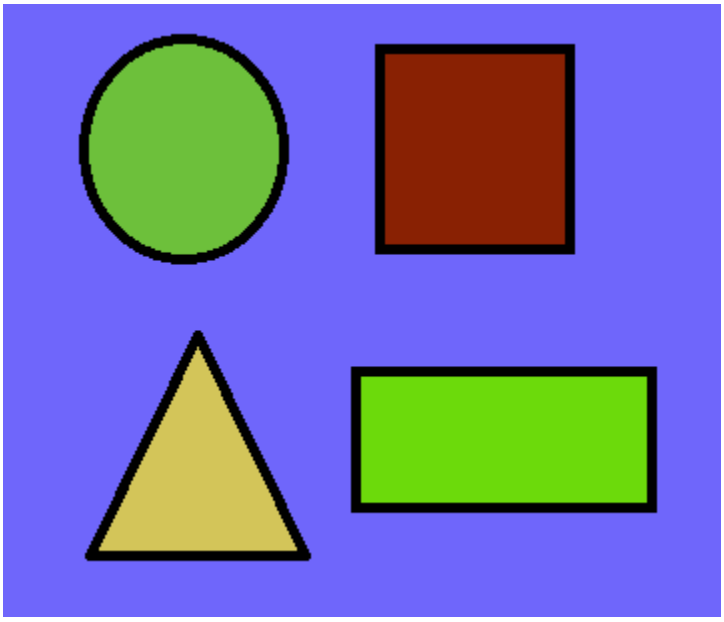
Identifying Foreground and Background (Figure 5)



To determine the foreground and background of the original image, I first converted the pixel's color from rgb to grayscale. On the grayscale, (255,255,255) is white and (0,0,0) is black. Therefore, if the average of the rgb values (grayscale of the pixel) was closer to white, I would assign the pixel to be white. If, on the other hand, it was closer to black, I would assign the pixel to be black. The resulting image is black (background) and white (foreground), no gray colors.

The foreground and background distinction follows the grayscale process with the addition of comparing the total to either white or black and making a decision. However, each pixel is still visited to complete the image transformation. The summations are constant time. The comparisons are constant time. An image with x width and y length will have (xy) pixels, where x and y are constants. Therefore, the runtime is O(xy). If we can find an n that is greater or equal to both x and y, the runtime can be written as O(n^2).

Component Detection (Figure 6)



After determining the foreground and background of the original image, I have to determine any connected paths. In other words, I have to detect the different objects in the foreground. Each object in the foreground is separated by a non-foreground object. In this case, the separation can only be found in the background which is all black. Therefore, I must check each pixel and see if it part of the foreground before contemplating any sort of change to the pixel. If the pixel is black, it is part of the background and so no change is needed. If the pixel is white, I decided that checking the pixel directly above and directly to left of the current pixel will indicate the necessary color of the current pixel. The intuition to this algorithm is that surrounding pixels indicate the color of the current pixel. If either the top or side pixel are black, they have no weight in deciding the color of the current pixel because they don't share the same ground. In the case in which the top and side pixel are both black, the current pixel is the start of a new object so a random color is generated for the pixel in the form of a Red, Green, and Blue value. There is a small adjustment for checking top right pixels in case both the top and side pixels are black. This further ensures that the current pixel is only a different color if it cannot find surround non-black pixels. However, nothing to the right on the same row or below the current pixel can be referenced with the way I decided to traverse the image. The whole process repeats for the next pixel.

Using this process, each pixel must be visited to at least check if it black or white. The comparisons with the top and side pixel are constant time. The small check with top right pixels has a constant number of loops, so another constant. Assigning colors are also constant time. An image with x width and y length will have (xy) pixels, where x and y are constants. Therefore, the runtime is O(xy). If we can find an n that is greater or equal to both x and y, the runtime can be written as O(n^2).

To execute the connected components algorithm, enter "node compDetect.js"

Process Scheduling

Process scheduling refers the order a number *n* must execute in order for the system to work. This is used when a process *w* might depend or use the output from process *v*. In this case, process *v* must happen before process *w*. A process can depend on more than one process as well. Therefore, the problem then becomes, if given a number *n* processes with *m* dependencies (paths) in an acyclic graph, what is the order than the *n* processes must execute in?

A solution to this problem is Kahn's Algorithm for Topological Sorting. Topological sorting is for directed acyclic graphs (DAGs) only. The DAG is represented as an adjacency list for this algorithm. Kahn's algorithm takes advantage of the fact that there exists a process that has no dependencies, or no paths into the node. First, the number of dependencies must be calculated. Any process or node with zero dependencies is added to a queue. A node is dequeued from the queue and added to another array to keep track of the order. If the node has outward paths leading to another node, that node's dependency is lowered by one. If that node's dependency reaches 0, it is added to the queue. The process then repeats for the next node in the queue, until the queue is empty. The array keeping track of the order can output the correct order the processes must execute after the queue is determined to be empty.

Setting the dependencies of each node if done in the algorithm means traversing every node and any paths from each node. This is run in O(|V| + |E|). However, I have the dependency as part of the creation of the graph so there is no separation of dependency initialization and graph initialization for my version.

The initial searching of nodes with no dependencies looks at each node. Therefore, it is in O(|V|). Adding the found nodes to the queue is constant time within the loop.

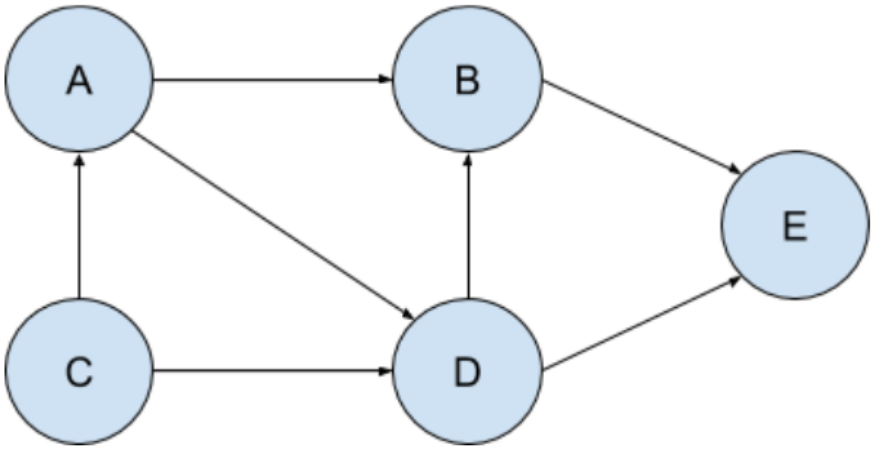
Dequeuing the first element in the queue happens in constant time. This element is saved in a variable. Next, we must search for that element in the adjacency list, so visiting every node in the worst-case scenario. Once a matching node is found, we must go into its list and traverse every outwards path and reduce the dependencies of the connected node(s) by one. If the node(s) dependency is zero, it is added to the queue in constant time. Therefore, the runtime is O(|V| + |E|) for the traversal.

Overall we have O(|V| + |E|) + O(|V|) + O(|V| + |E|). This can reduce to O(|V|)+ 2O(|V| + |E|), but O(|V|) is less than 2O(|V| + |E|) and the constant can be dropped. Kahn's algorithm has a runtime of O(|V| + |E|).

To execute the process scheduling algorithm, enter "node orderProcess.js"

Below are the two examples used as test data. Each figure is accompanied with the correct and necessary order of execution.

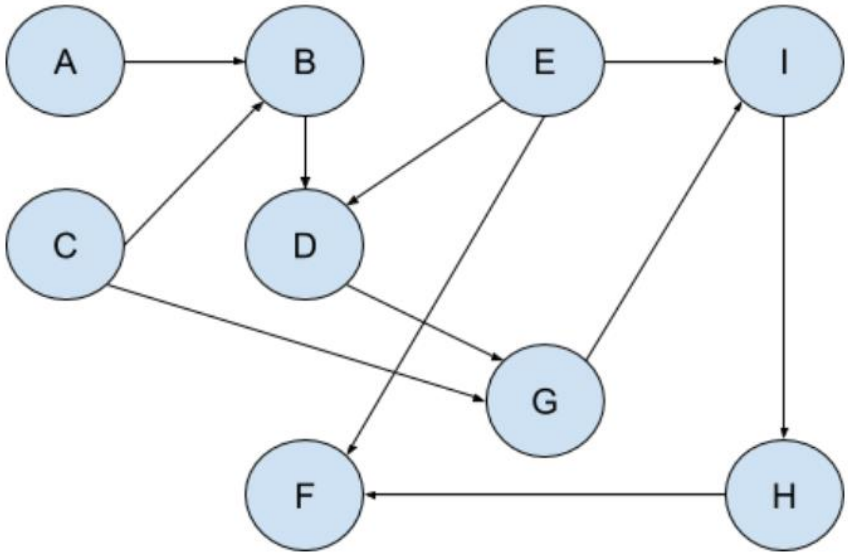
Graph 1 (Figure 7)



Order of execution

C → A → D → B → E

Graph 2 (Figure 8)



Order of execution

A → C → E → B → D → G → I → H → F

Summary of algorithms executions

Image Manipulation

Input is in the source code. For now, it is "shapes.png" The output saves as the same file type as the input. A different input image can be used as explained in the source code.

Grayscale	node grayscale.js	output image: "imageGray"
Blur	node blur.js	output image: "imageBlur"
Sharpen	node sharpen.js	output image: "imageSharp"
Connect Components	node compDetect.js	output image: "imageGroundDetect"
		Final output image: "imageCompDetect"

Process Scheduling

By default, both figures 7 and 8 (from above) are put through the algorithm. The terminal will display the ordering for both graphs, individually.

Process Scheduling	node orderProcess.js	output: on terminal

To execute all, enter below line (all output are same as shown above):

node grayscale.js | node blur.js | node sharpen.js | node compDetect.js | node orderProcess.js