All runtime complexities will be calculated in terms of the width (w) and height (h) of the input images, and the number of seams to be removed (k).

Image to energies:
Both functions first turn an image into energy values. They do this by sequentially calling the functions "image-to-brightness" and then "brightness-to-energy".

<u>Image-to-brightness</u> gets pixels of an image which is linear in the product of w and h (given in assignment), and the width and height of the pixel list which takes linear time in w + h. It then maps over each pixel in the image which is linear in the product of w and h. It then adds a 0 to the front and back of each row in the image. This is linear in w for each row (since append is linear). It runs for each column, making it linear in the product of w and h. Finally, it adds a row of zeros of length w + 2 to the front and back of the list of rows of the image. Making the row of zeros is linear in w, and appending it to the back is linear in h (while adding it to the front is constant), making this linear in w + h. Since each of these is run sequentially, we take the worst time complexity, **giving image-to-brightness a time complexity of O([w,h → wh]).**

<u>Brightness-to-energy</u> runs a function called "energy-helper" on the first three lists in the 2-d list of brightnesses it inputs (which has length h + 2, and each list within it has length w + 2). It then recurses by running brightness-to-energy on the rest of the main list (so stripping one list from the 2-d list) until there are no longer three lists to run on. This recursion is linear in h since it runs once for each list in the main list.
Energy-helper first gets the first three elements of each of the 3 lists passed to it using .get. Since it does this strictly for the first three elements (not the whole list), this takes constant time. It then does some math calculations, which is also constant time. Finally, it makes a recursive call on the rest of each list. Since it runs a constant cost for each item in the input lists, it is linear in the length of the lists, which is w + 2, making it linear in w.
Since brightness-to-energy runs energy-helper for each item in the list, we multiply the runtimes, meaning **brightness to energy has a time complexity of O([w, h → wh]).**

Since image-to-brightness and brightness-to-energy run sequentially, we take the worst runtime of the two. This means that **converting images to energies has a runtime of O([w, h → wh])**

Memoization:
The memoization function then runs a function called "min-seam" on the 2-d list of energies.

Min-seam consumes a 2-d list of energies and produces a list of numbers representing the indexes of the min seam. It first gets the dimensions of the energy list which is linear in w + h. It then memoizes a function which just immediately returns a function which takes constant time. It then calls min-seam-helper.
Min-seam-helper goes through each list in the 2-d energy list, which is linear in h.
Min-seam-helper calls calc-min-seam-memoized for each item in the first list, and 2-3 items in the second list.

Calc-min-seam-memoized consumes a list of list of numbers representing energy and 2 numbers representing a row and column index, and returns the min cost of a seam going from that point in the list to the bottom of the list. will run on each pixel in the input image once because it is memoized (and remaining times it will just access the stored value). This means that we can calculate the runtime of it on all pixels and then treat each call to it as a constant cost that just accesses the memo-table. Each call in the function is constant besides the call to get a specific index of the 2-d list. Gets are linear, which means that getting an item in a 2-d array of size w, h is linear in wh. This is run for each pixel, so wh times, giving a runtime of (wh)^2. While technically these calls are made at different times, calc-min-seam-memoized is called once for each pixel, so we can isolate that runtime calculation instead of saying that we have a general runtime for the function of (wh)^2 run w times by min-seam-helper.

All other calls in min-seam-helper are constant. Rather than calculate the runtime for min-seam-helper individually, this analysis tells us that the runtime over all calls that min-seam makes to min-seam-helper recursively is (wh)^2. We take the worst time complexity out of everything the function runs separately and ignore the linear calls at the top, meaning **the time complexity for min-seam is O[(w, h → w^2h^2)]**.

The memoization function then gets the pixels from the image and its dimensions, which has a total time complexity of O[(w, h → wh)] (which was calculated above). Finally, it runs map2 on the pixels and list of numbers representing the indexes of the min seam. The function inside the map 2 calls take and drop, each of which are linear in the elements in the list they take and get from, making them linear together as well. There are w elements in each list, and h lists, **making the map2 operation O[(w, h → wh)].**

Since each of these things are run separately, we take the worst time complexity, giving each iteration of liquify-memoization a runtime complexity of O[(w, h → (wh)^2)]. This function is recursively run n times, giving a final time complexity of **O[(w, h, k → w^2h^2k)]**.

On each iteration, removing a min seam has a set runtime that isn't affected by the nature of the pixels provided (for example, it doesn't matter what color the pixels are or what the dimensions are absent the total number of pixels). The input n is directly multiplied by the rest of the runtime no matter what, meaning that the value of n won't affect if you get the worst case runtime either.

This means that **the runtime is essentially set and not based on the nature of the input, it will run in O[(w, h, k $\rightarrow$ w^2h^2k)] every time.**

Dynamic Programming:
All of the functions in the dp version of the assignment are identical besides “min-seam”, which is replaced by “calc-min-seam-dp” and “get-min-seam”. This is the dp version of the function.

Calc-min-seam-dp consumes a list of list of numbers representing energy and produces a list of lists of numbers representing the min cost of a seam from each pixel to the bottom of the image. It first finds the number of rows and columns by calling length which is linear in w + h. It then gets the last row, which is linear in h as well. It then builds a 2-d array with the dimensions of the input list of lists. This most likely (because it is a built) incurs a constant cost for each item placed in the array, which would make this linear in wh. It then operates on each item in the 2-d array. It gets each row of the array and sets each item in each row. Getting and setting is constant time for arrays, so this all is linear in wh (since those are the dimensions of the array). It also gets each list in the input list of lists and each item in each list a constant number of times. This has a runtime of $O[(w, h \to w^2h^2)]$ (which was calculated earlier). Finally, it converts the array into a list of lists. This most likely is linear in wh since it incurs a constant cost for linking each item in the array into a list, and there are h arrays each with w items. Since all of these are run sequentially we take the worst runtime, which is **$O[(w, h \to w^2h^2)]$**.

This is the same as the runtime for the memoized version of the operation, giving **a total runtime of $O[(w, h, n \to w^2h^2n)]$.** Once again, **the runtime is essentially set and not based on the nature of the input, so it will run in $O[(w, h, n \to w^2h^2n)]$ every time.**