Stephen Norod: Code-Foo 2017 Application

1 – Video

Here's the link to my video https://www.youtube.com/watch?v=cC4RzPkhkYQ

There's an annotation in the video linking to my website (snorod.com) for some follow-up information.

2 – King Kong

It would take 289,546,164 Lincoln Logs to replicate the Empire State Building. My thought process began with the idea that the structure can't be hollow on the inside, because then if King Kong climbed it, it would just fall over. Then I realized, even if it was entirely solid, there's no way that it could sustain the weight of King Kong. Based on the facts that a typical ape of about 5/6 ft tall weighs about 250 kg, and that Kong is typically depicted as about 50 ft tall, Kong's weight can be estimated to be 2500 kg or about 5500 lbs. Given that wood is nowhere near dense enough to sustain this weight regardless of its structure, I restarted the problem under the assumption that the Lincoln Logs are conjoined with really, really, really sticky glue.

So, with that in mind, the most cost effective solution would be to only provide enough Lincoln Logs to cover the surface area of the Empire State Building, but that's no fun; Kong probably wants the most realistic version of his favorite building. So, I decided that enough Lincoln Logs should be provided to cover the entire surface area of the building as well as enough to act as floors/ceilings inside the building. Under the glue assumption, it doesn't matter how thick each floor should be, so the most efficient method would just be to have a thickness of one Lincoln Log. Then, there are the windows. There can't be any Lincoln Logs where a window should be in the ESB. Plus, if there are empty spaces where the windows should be, Kong could look into the building and see my handiwork as well as have helpful finger holes when he's climbing.

Lincoln Logs are famous for their locking mechanism, but since it is already established that the Lincoln Logs will not withstand Kong's weight without the help of glue, this locking mechanism becomes arbitrary and it would be less efficient to utilize it. Instead, it would be best to simply conjoin each Log end to end, thus maximizing the total area a group of Logs can cover. The ESB is, according to the Internet, 87,120 ft^2 in surface area (with spire accounted for, I believe) and 37 million ft^3 in volume. Lincoln Logs come in various sizes, which is an issue to be considered. The most wood-efficient Lincoln Log size is the smallest available one; each Lincoln Log has indented areas where other logs are supposed to latch onto (locking mechanism) and the smallest Lincoln Log piece is essentially just the locking mechanism itself, meaning the entire body of the piece is indented. This piece is, also according to the Internet, 1.5 by 0.75 in, or 0.125 by 0.0625 ft. The surface area of this piece (disregarding its indentation) is therefore 0.0078125 ft^2. If the surface area of the ESB is divided by this, the result is 11,151,360 tiny Lincoln Logs to cover the outside.

Now this quotient must be added to Lincoln Logs needed for each floor/ceiling. The total floor area of the ESB is 2,248,355 ft^2, so we can divide this by the previously found 0.0078125 ft^2 of a Lincoln Log. The result is 287,789,440 Logs. If this is added to our previous answer, we get 298,940,800

Lincoln Logs! Nearly 300 million little pieces of wood, plus some glue. A few additional assumptions were made here that should be noted. First, the accumulated additional volume of the glue between the logs is negligible, second, additional layers of Lincoln Logs on each floor and wall (for reinforcement) are not necessary due to the glue, and third, the glue is weather/temperature resistant.

There is one step missing here, which is the subtraction of Lincoln Logs where there should be windows. There are 6,514 double-hung windows in the ESB. An exact estimate of each window's area could not be found, so an average size of 55 by 29.5 in was chosen. The surface area is therefore 1,622.5 in^2. Dividing this by the surface area of a Log (1.125 in^2) results in 9,394,636 Logs for all windows. Subtracting this from the previous answer yields the final result: 289,546,164 Lincoln Logs.

3 – Boggle

To run: Within the Boggle directory, run "javac BoggleGame.java" and then "java BoggleGame"

This design process has been a very long, but enjoyable, series of epiphanies. I basically completed the problem twice, using two completely different methods. I'm going to explain the important decisions that I made, including why I decided to start over and why this was a good idea.

I've made games in Java before, so I decided early-on that I would use this language again. My first realization was that it would be extremely helpful to make each tile on the board its own object with x and y coordinates as well as a value; this way I could compare them to each other when checking for solutions.

I hit a roadblock when trying to figure out how to find all the solutions. This is where I made a poor decision on my first try and where I came back to on my second try. I wanted to make a sort of AI that could find all the solutions, but couldn't come up with a way to execute this. At some point, it hit me that I could just compute every single combination of the numbers on the board, and then screen these based on the requirements of having adjacent cells, adding up to the area of the board, not repeating cells, etc. This approach works for a 3x3, but not if you scale the program up (as I discovered later on).

One major bug I ran into is when I check if cells are adjacent to each other. My original method was just to check if each cell in a chain is connected to at least one other tile. This is fine, except for when you have 4 or more cells in the chain. For example, when checking the numbers 3420, the 3 and 4 could be adjacent to each other, but not near the 2 and 0 which are adjacent to each other. So, the program would say that 3 is attached to 4, 4 to 3, 2 to 0, and 0 to 2, thinking this is a viable path based on the fact that each cell is connected to at least one other cell. I realized that another way to check if a chain is viable is that the number of cells that must have at least TWO connections is the length of the chain minus two (minus two, because the endpoints only need one connection, but every other tile must be connected on two ends). So, I then created an int array of the number of connections each tile in a chain has, and make sure at least size() – 2 elements are greater than or equal to 2, and that none of the elements are zero. If none of that made sense, it doesn't matter, because it all changed anyway.

At this point, I thought I was done and focused on the visuals for a bit. From the beginning, I wanted to make a JavaFX GUI, just because I think they're nicely minimalistic and I have some experience with them. One major component that I thought would be cool, and that I knew would also

be helpful for debugging, is highlighting each solution path on the board. I did this very carefully and it surprisingly worked; I'm most proud of myself for making this aspect of the program, because it was difficult to map a string with "3+6=9" to the corresponding buttons on the physical board.

After finishing the GUI, I noticed that some long solution chains were not being screened properly, an issue I thought I had fixed earlier. This is a recurring problem. I'm also becoming wary of all these screening tests I'm adding in to account for the 4x4, because I don't know if they are affecting my already-working 3x3 (they were).

I restarted the program and decided to model it after the depth-first search algorithm. This is for navigating all the paths in a tree. I figured that I could treat each tile on the board as the root of its own tree, and recursively traverse the possible paths leading out of it, using sum == area as an end case. Each tree is non-binary, so there are more than two ways to go at each point, which means that a loop is necessary for iterating through each root's children. This method significantly increases the efficiency of my program, because when the program knows a path won't lead to a solution, it won't continue on that path and won't need to calculate any further. Now, I can actually do a 5x5 (although this should generally be avoided due to the amount of computation required), whereas my first program froze every time I tried. I think it would scale easier if the range of numbers (0, 9) was also scaled, because the 2x2 hardly ever has a solution and the 4x4 always has many solutions, but for now I'm very happy with my program.

4 – App

To run: Go to https://facebook.github.io/react-native/docs/tutorial.html where there is an emulator that supports React Native, copy/paste the code from index.android.js here. Note, you would usually run on an actual Android Studio emulator, but without React Native installed on your system, you wouldn't be able to do that. An APK can also be generated.

I took a risk here and I mostly regret it, because I didn't finish. I knew I wanted to do the Android app, but I wanted to make it a little different. So, I decided to try React Native, something I'd never done before. I've worked on Android multiple times of course, but React Native is very different, because although it makes a real Android app, it is written almost entirely in JavaScript. A huge benefit of this is that the code can be reused to make the iOS app, bridging the gap between the two platforms. I thought to myself that it was worth trying, to show that I was willing to adapt to new technologies, but I ultimately ran out of time to complete the challenge. By the time I realized that I wasn't going to be able to finish, it was too late to try starting over in Android Studio.

At the beginning, I looked at the official IGN app and tried to think of what was worth replicating for my cheap imitation. I decided that the search icon would be worthless, but the ability to save an article so it wouldn't disappear after the next API call would be nice. I chose to keep the saved list feature as well as the top-left menu icon. The menu would just be used to show "by: stephen norod" or something like that.

Something I wish I could've implemented is how, in the real IGN app, the bar at the top disappears when you scroll down. Mine is unfortunately static. I also planned to have a Navigator element running behind everything, so that the saved list icon and menu (info) icon would each be

buttons leading to their own pages (although they do highlight when you click on them now, they don't do anything). As for the API calls, I've never used the Fetch API before and ran into some trouble with it, obviously. I definitely shouldn't have saved that part for last.

I hope that my effort to do something out of the box with this app somehow counterbalances my failure to make it work. When it comes down to it, I underestimated the amount of time it would take for me to adapt to React Native. With more time, I would probably start over with Android Studio just to get a reliable product.

Bonus – Qwirkle

To run: Within the Qwirkle directory, run "javac QwirkleGame.java" and then "java QwirkleGame"

The only reason I included this program in my submission is to show that I at least tried it and that I was able to get as far as instantiating the board as well as the player's and cpu's hands. If you run this program, you will quickly discover that it looks similar to my Boggle game and also does not work. I tried making this in Scala Swing just to mix things up a bit, but I didn't like it. So, I figured I would try to use my Boggle game as a foundation and work from there. I don't think completing this would've been particularly difficult, just time consuming. However, I wanted to implement a drag-and-drop feature for placing tiles on the board, which I definitely think would've been a challenge.

Closing Thoughts

At this point, I'd like to say that I know I didn't technically finish my application, since my app doesn't work how it should. With more time, I'm positive that I could've adapted better to React Native or done a satisfactory job in Android Studio. Code-Foo took a bit of a back seat at times in favor of work that I actually NEED to do, but if I didn't have to worry about other work, I think I would've had a much better application. I hope that IGN will still consider me as an applicant, because I really am passionate about IGN and I love programming. I can do better than the app I'm submitting and I want the chance to show that at Code-Foo. Regardless of your decision, thank you for the opportunity.