Project #3

Daniel Williamson

11/13/16

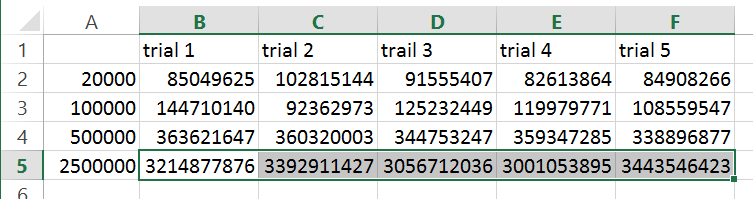
MergeSort: O(log(n))

Approach:

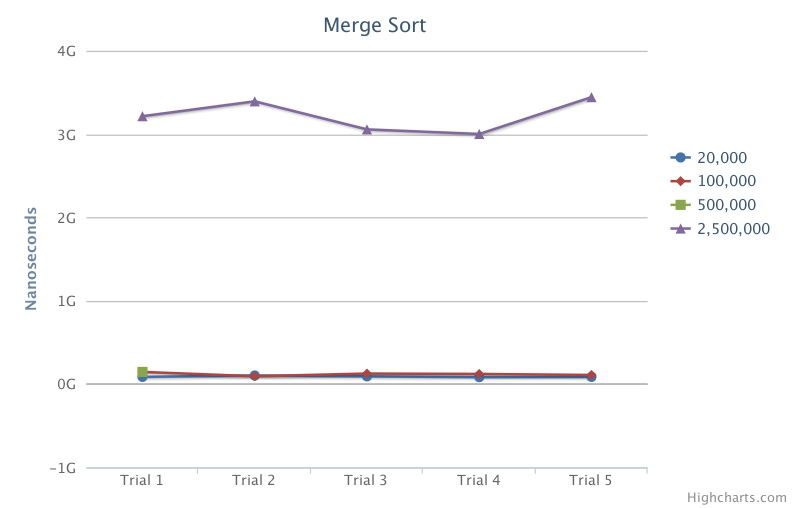
My approach for this project was just like many others. I start by doing a lot of research. I reference the book, the internet and peers. Stephen White was one in specific that that was a giant help. After figuring out how MergeSort worked. It was a rather simple concept. Basically, it takes arrays and reduces them in size with recursion, tell it hits our base case of 1 in size. Which once hit, it shoots back up, correctly placing each numerical value in the correct order.

To begin, I needed to understand how the skeleton code was working. There were three classes in it, sort, which passed in just a list. Another sort, which passed in a list and comparator. And then lastly the mergeSortRecursive, where all the work happened. In the first Sort, where a list is passed in, we have to essentially create a comparator for our other sort class. In case the user just passes in just a list. Once we have a comparator created, we send it on over to the second sort method which just calls mergeSortRecursive. To handle the mergeSort we needed to create a helper method, which I called just merge. Which, simple just merges the left lists and right lists that the mergeSortRecursive creates. While mergeSortRecursive goes through, it creates two sides, a left and a right. Which our helper method sorts. Now, that was simple it, however, you guys created a test, the internal calls, that doesn’t want to start from the beginning and start somewhere further in, and it doesn’t want to end at the end. So, in order to handle this kind of problem, we created two more helper methods that were rather simple. SetterFrom and setterTo. Which sets from where to begin and to where it ends. This lets us set our values so we know where to start and where to end. And then in our mergeSortRecursive we have an, if statement that handles the internal call. If the start does not start at 0 and the end does not go to list.size. Then we know we are doing an internal call. And then we set the from and to values, and go on our way.

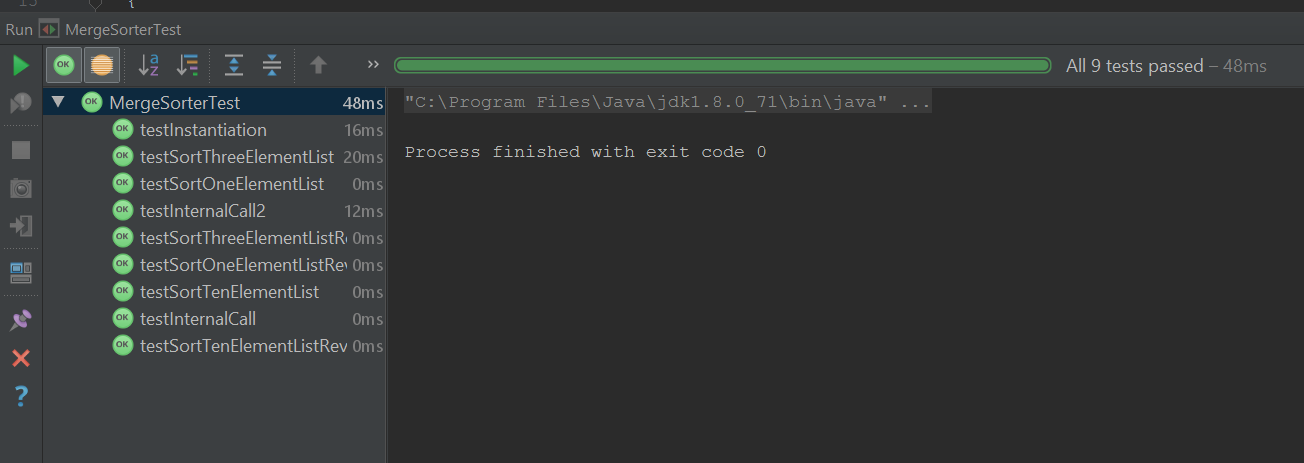
After all tests had passed, it was onto creating the main method to print out the time it took to sort certain sizes. Here are our findings:



And in a chart, now sorting the 2,500,000 took so long, that it kind of makes all the others look so small.



Here is a photo of the code passing all the tests:



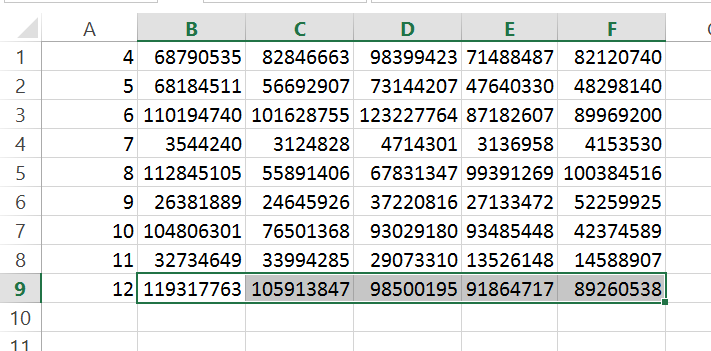
nQueens: O(N!)

Approach:

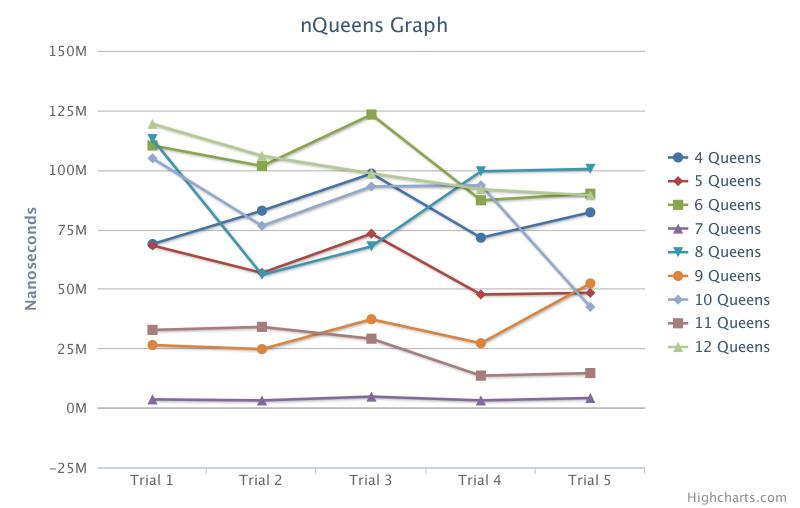
Just like I said, I started nQueens with doing some research on how it needs to work. I used the internet, the book and peers. Namely Stephen White. nQueens was rather easy to understand what needed to be done. But making that happen was rather difficult. Essentially, we have n amount of queens, and a board set at the same size of number of queens. We then must place all these queens on the board, with all of them being “safe”. Being safe implies that no two queens are horizontal, vertical or diagonally from each other. So, we use recursion to solve this puzzle, by constantly failing, we learn that, that is not a solution to the puzzle. We keep failing and solving tell eventually we hit the solution.

The idea is rather simple, but to solve this, was really difficult. We start by filling in the nQueens method, which just returns nQueenRecursive where all the work happens. In order to get nQueenRecursive to work properly, we had to create some helper methods. canDo, is a method that checks to see, if we can place a queen at a spot. It will check rows, columns and diagonals. If any one of them can’t be done it will return false, if all of them pass, it will return true. This is important because we use this in nQueenRecursive which canDo is called every time we try to place a queen on the bored. So, for nQueenRecursive, we take in an int n, a prior Boolean 2D array and two ints col and row. So basically, we place a queen on the bored in the first column. We then move over a column and try a row. If it works, we go to the next column. And so forth. If we place a queen that can’t be place, we try that again, in a different row. If we get to a point where we can’t place a queen and we still have some left, we got to do some backtracking, and we try a different solution. Over and over again we try this, tell we get the correct solution.

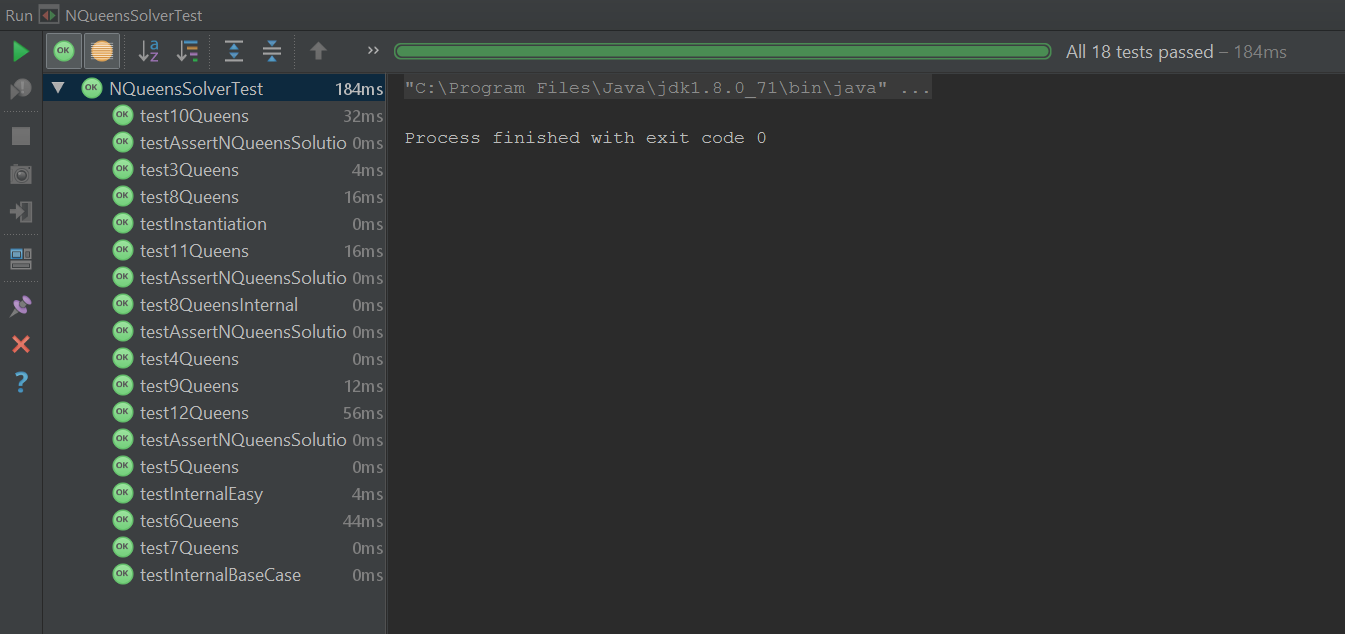
After we passed all the tests, we created our main method to print out, in nanoseconds, the amount of time it took to solve the puzzle with certain amount of queens. Here are our times:



Here is a chart with those values, surprisingly 7 queens was solved the fastest:



Here is a photo of the code passing:



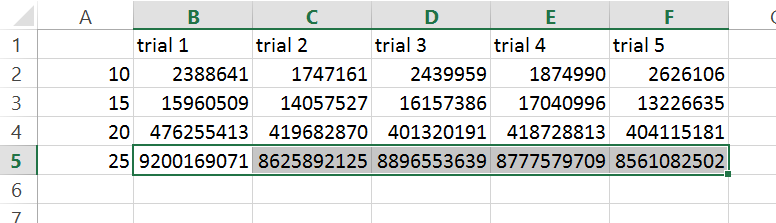
Knapsack: O(2^n)

Approach:

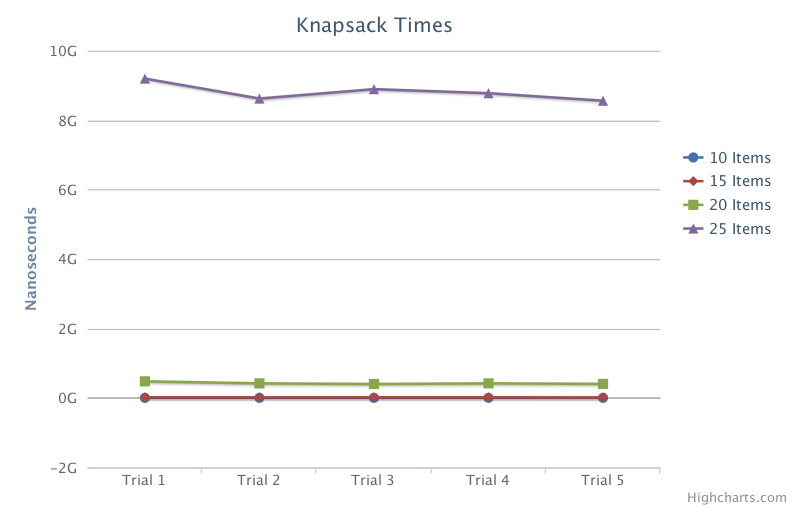
This was, without a doubt the most difficult part of project #3. I conferred with the internet, my book and peers. Once again, namely Stephen White. We even approached Mr. Kelley about this project. The idea, is once again, rather simple. It essentially is a sack/backpack. It holds a certain amount of weight and we have an array of items, each item has its own weight, we place items into this backpack in hopes to meet the predetermined size of the backpack. If we exceed this size, we try again. If we have less then, we try shoving more in. If, eventually we hit the value of the backpack we have been successful.

To begin, we filled out the knapsack method, which we just create a Boolean array and return the capacity of the bag, the items we have to put in the array, and a Boolean array named prior. We are using a Boolean array, because if an item won’t work, we make it false, if an item does work in our sack, we make it as true. And then we can return the Boolean array to see if we have the correct Boolean values and can compare them easily. So, to get knapsackRecursive to work properly we had to create two helper methods. One to get the weight of the bag and return with it. Another to get the value of the bag and return it. To begin the knapsackRecursive method we start with our base case. If prior.lenght eguals items.lenght we return prior, meaning the puzzle had been solved. We then create two, temporary values, that may or may not be a solution. We then fill those arrays with whatever prior was. We then create another two arrays for after the recursion. We also make values and set them to -1. Incase anything happens. Next, we check to see if we can put anything into the bag. Using recursion. If the value of the bag is less than or equal to the other value, we know this might be closer to the solution.

Once we passed all the tests we created our main method to run and time how long it takes to solve the puzzle. Here are our times in nanoseconds:



And here are those values in a chart. Needing 25 items took the longest:



Here are all the tests being passed:

