11/10/2016

Project 3: Recursion

CS249

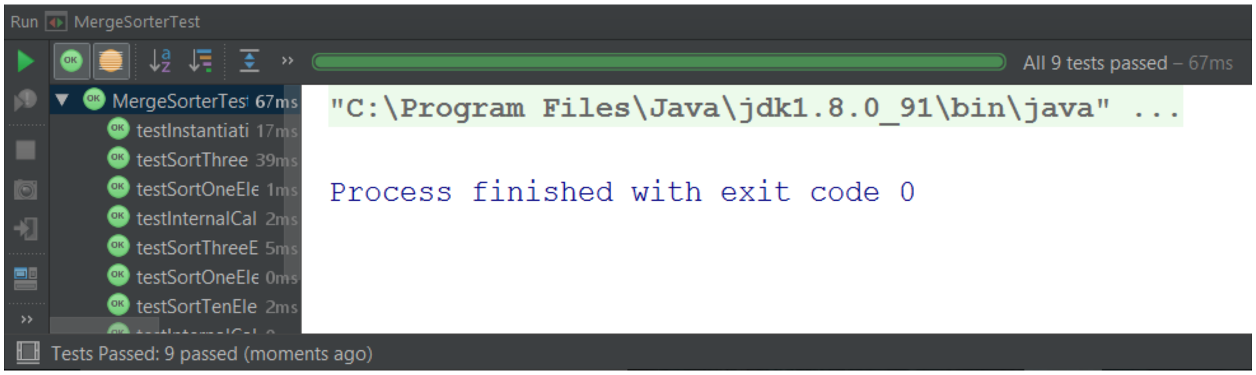
Instructor: Patrick Kelley

Stephen White

Overview: This project was purely about utilizing recursion as a "divide and conquer" method to solve very complex problems in computer science. There were three separate recursive programs that I had to implement for this project: Merge Sort, the N Queens Problem, and the Knapsack Problem, with each taking its own twist on how I should go about solving them. The road to completion was a nightmare, but let's take a look at how I got here.

Process:

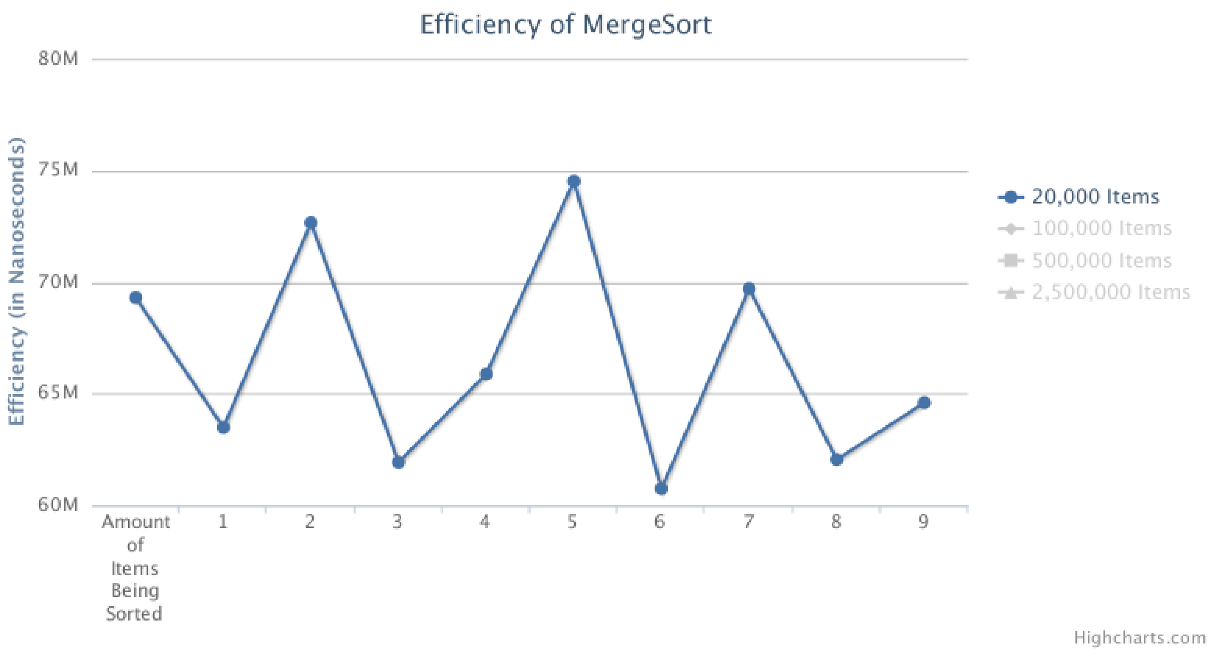
Merge Sort: My process began with me working on Merge Sort. My approach to this portion of the project and all other portions of the project was nearly identical. To understand each recursive program, I created my own "playground.java" of sorts to break things down as simple as I could and work up to the final program from there. For Merge Sort, this meant that I would try to get the program up and running without utilizing any comparators or generics first, then once I got the sorting to work properly on just Integers, I would begin to alter code to make my way up to the tests I needed to pass. To be able to properly sort items from a list, I decided to go about writing my own merge method. This would require that the user pass in a "left list" a "right list" and the "original list," as well as a comparator in the long run. This method essentially utilized index pointers and while loops to compare items in both the left and right lists, would find whichever item was bigger, and *overwrite* the values of the original array. This in turn, allowed my method to not have a return type, and run at approximately O(N) efficiency. My next step after correctly designing the recursive algorithm was to make my program generic, and to my surprise, this took little to no effort at all, and I was able to get it done pretty quickly. The last little hurdle came to me in the form of implementing a comparator, which in itself was not too incredibly difficult, and required only a few tweaks to get right. At this point, I had not yet tried to test my program with the unit tests, and was surprised that the only thing I was failing were the two internal calls that took in a list, and sorted only a small portion of the list with the "from' and "to" variables changed. *This,* was when I started really pulling my hair out, as well as when I learned that the integer that I was returning to the user was not correct. Everything was being sorted correctly, but what was it that I was counting!? After about a whole day of being stumped, I decided to test my theory by incrementing a count with the size of the list being passed to each recursive call and presto! I only had one more fight to finish off this recursive program and that was the internal call! I asked myself, "how would I be able to determine if someone was trying to access my recursive algorithm directly, and be able to handle it from there?" The answer to my question: a massive if statement. I came up with the following conditional: if(count == 0 && from !=0 && to != list.size()) → handle internal call. You see, if I knew that nothing had been sorted yet (i.e. my count was equal to zero) and my "from" and "to" variables were not handling the list as a whole (0 – list.size()) than someone was trying to sort only a portion of a list, and I would need to handle that appropriately. How did I handle this, you ask? I kept track of what the original from and to variables were, created a copy list based on the original list from "from" to "to," split that list in half, called the recursive method on both the left and right lists, merged them back into one list, and at the end, overwrote the values in my original list from "from" to "to" using the values of my sorted copy list, and it worked like a charm. After successfully passing all of the tests, as you can see below, I wrote a class that's sole purpose in life was to time how long it took Merge Sort to sort 20,000 items, 100,000 items, 500,000 items, and 2,500,000 items, and created a line graph for your pleasure. Shown below is a screenshot of all unit tests passing.



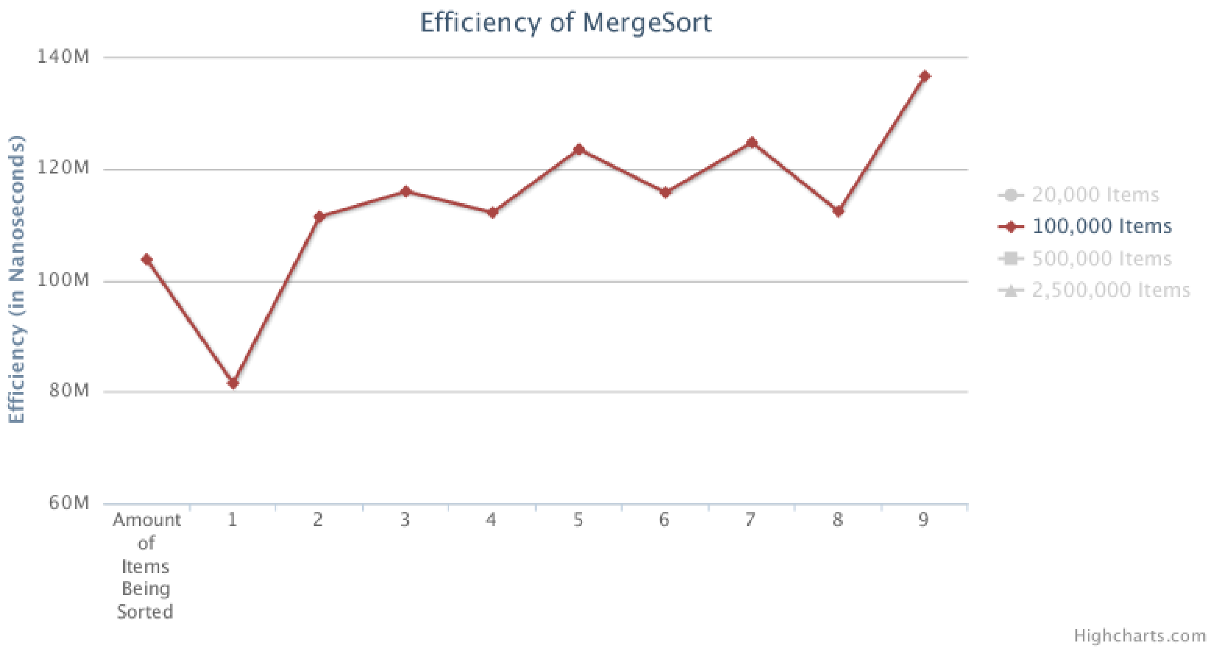
```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...

Process finished with exit code 0
```

Tests Passed: 9 passed (moments ago)

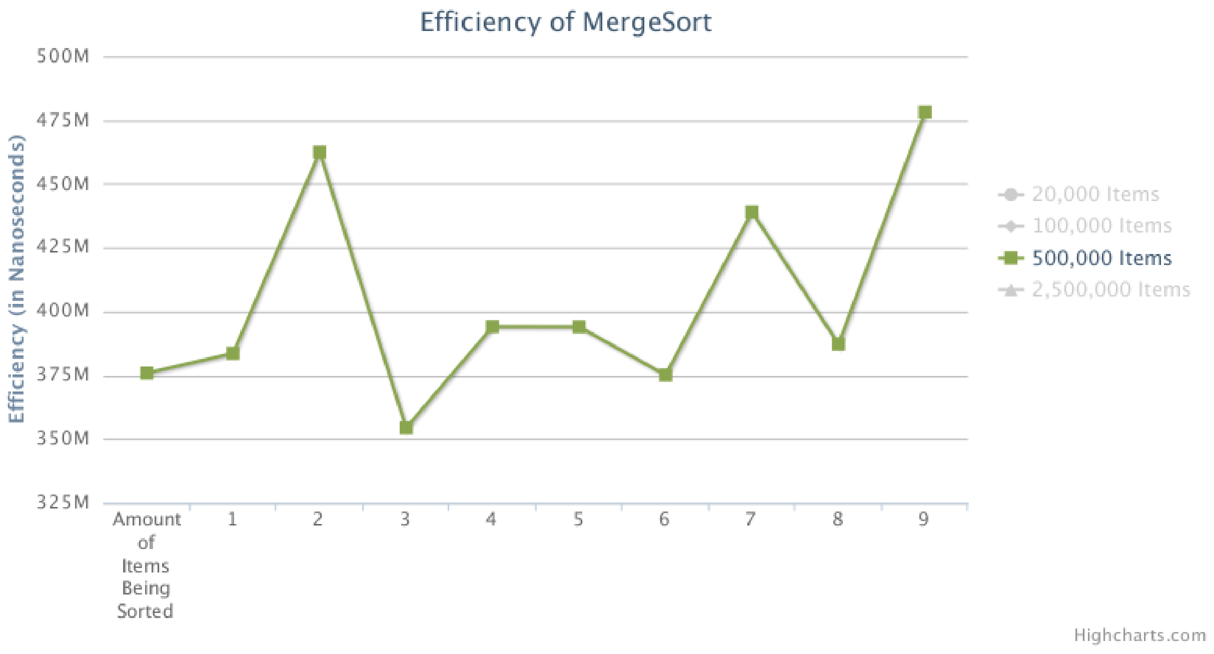
***NOTE: In the following graphs, the X-axis should represent a separate test, not the amount of items being sorted.**



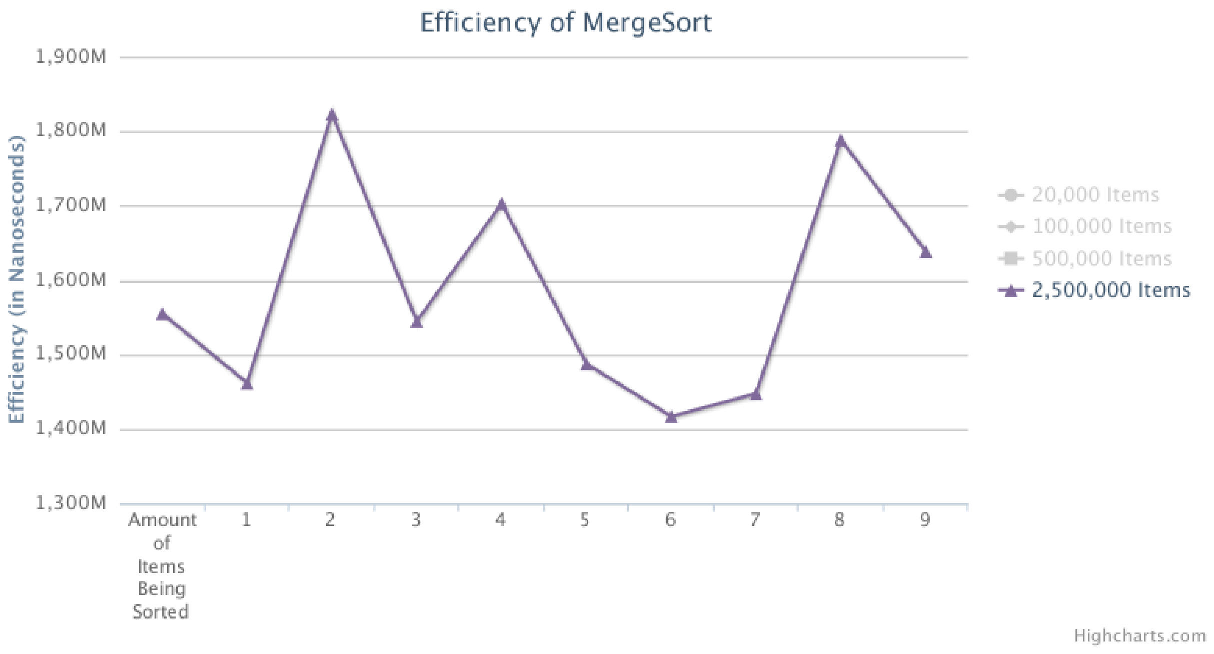
Efficiency of Merge Sort on 20,000 items (above).



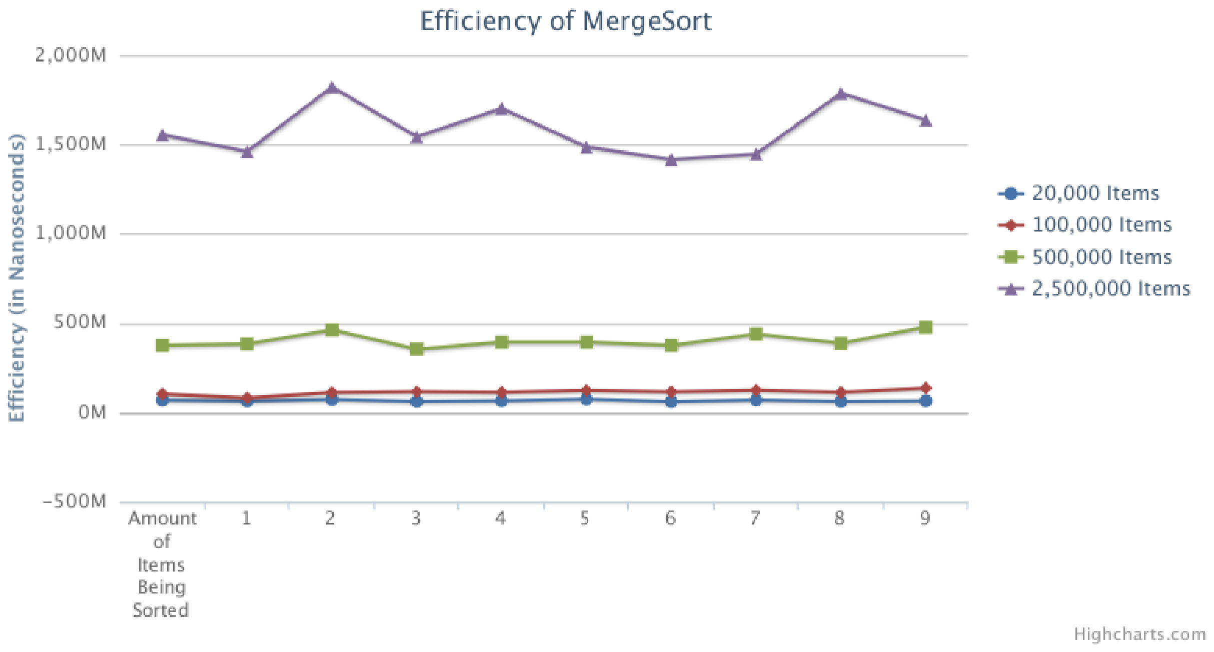
Efficiency of Merge Sort on 100,000 items (above).



Efficiency of Merge Sort on 500,000 items (above).



Efficiency of Merge Sort on 2,500,000 items (above).

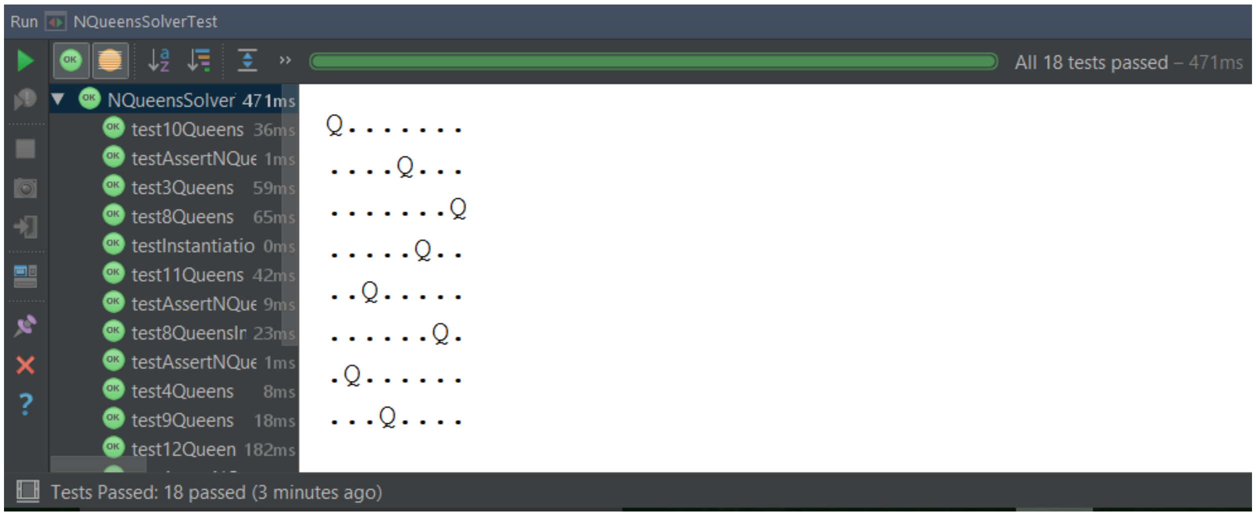


Full graph demonstrating the efficiency of Merge Sort (above).

Analysis of Merge Sort: As you can see from the above graphs, Merge Sort is capable of sorting a great number of things in a very small amount of time, with the delay in completing the sort ranging from 130121435 nanoseconds (approximately 100 milliseconds) up to 242519391 nanoseconds (approximately 250 milliseconds) on the low end of the spectrum, 665213152 nanoseconds (666 milliseconds) as a median for 500,000 items, and 2871822203 nanoseconds (2872 milliseconds) on the highest end of the spectrum. The more items to sort through, the longer it would take the program to finish. All in all, the efficiency of Merge Sort is roughly O(N * Log(N)).

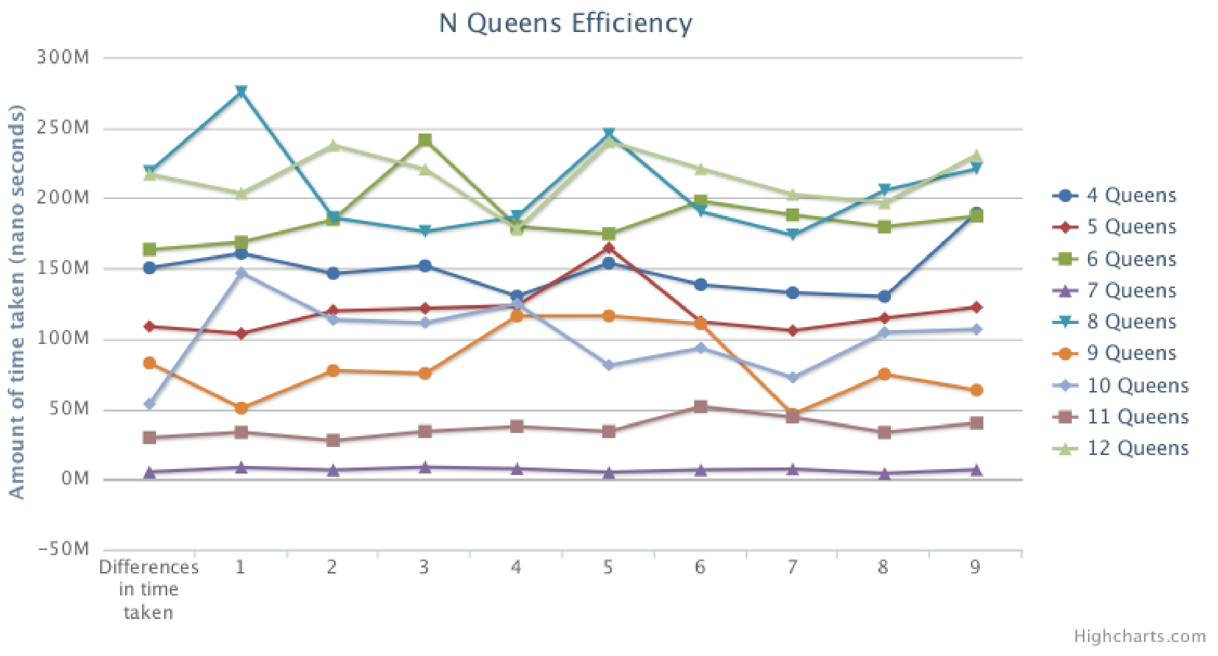
N Queens Problem: This portion of the project was an entirely new beast in itself. The N Queens problem required me to think in the form of a two dimensional array of Booleans, with true values representing the placement of a queen and false representing an empty space on a chessboard. The idea is that on an N x N chessboard, is it possible to place N number of queens on said chessboard without any one queen being able to kill another? If it is possible, I needed to return the actual representation of the chessboard, and if not, I needed to return null. I decided to work with a fellow classmate on this portion of the project, considering it took so much out-of-the-box thinking that it would have been nearly impossible to do on my own. I once again created a separate “NQueensPlayground.java” class to try and mess around with some ideas before I fully committed myself to trying to pass any unit tests. The first thought that came to my mind, while Connor Schwirian delved into the conceptual stress of the recursive part of the algorithm, was “how do I determine if I am able to even place a queen on the board at a given location?” With this in mind, I did what I do best, and developed several helper methods that I chained together that allowed me to check very cardinal direction from a point on a chess board, as well as all diagonals. When I had developed the method, I decided to name it isSafe() and it would require the user to pass in the current two-dimensional Boolean array, as well as a row and column. It would access the current two-dimensional Boolean array, and check every possible direction from the row and column given, and return a Boolean value of true if it was safe to place a queen here, and false if it was not. With that in mind, Connor had worked for about two days and had finally come up with a way for us to merge our code together and create something that worked. Here is how the final solution turned out: upon each recursive call, a count would determine how many queens were present on the board, and if the number of queens tallied by that count was equal to N, then we know we have solved the problem and would return the board as is. That was our base case. From there, we would reset the count, and determine if any queens were already present in the current column the recursive call was taking a look at. The most important step was the creation of a "checkpoint," or a "working copy" of the Boolean array that we were passed into the recursive call. This would allow me to backtrack if necessary and try other possible combinations. I then created a for loop that would iterate through the entirety of the chess board and ask the question: "can I place a queen here?" If I could, I would create a new version of my working copy, throw all of the values of the prior Boolean array into it, place the queen, and call the recursive method again, but set my checkpoint equal to that call. If I did not run into any problems (hence I would not have returned null) I would return the solution to the user. In the event that I was not able to find a solution with the current setup of queens on the board, I would return null, and essentially "backtrack" into a prior recursion, and try again with different values for row and column. Shown below is a picture of all unit tests passing, as well as a single graph that compares the run times for solving N Queens (between 4-12 queens) one hundred times per plotted point.

NOTE: In the graph shown below, the X-axis should represent the number of tests taking place, and not the "Differences in time taken."



```
Run  NQueensSolverTest
All 18 tests passed – 471ms
NQueensSolver 471ms
  test10Queens 36ms
  testAssertNQue 1ms
  test3Queens 59ms
  test8Queens 65ms
  testInstantiatio 0ms
  test11Queens 42ms
  testAssertNQue 9ms
  test8QueensIr 23ms
  testAssertNQue 1ms
  test4Queens 8ms
  test9Queens 18ms
  test12Queen 182ms

Q.......
....Q...
.......Q
.....Q..
..Q.....
......Q.
.Q......
...Q....

Tests Passed: 18 passed (3 minutes ago)
```

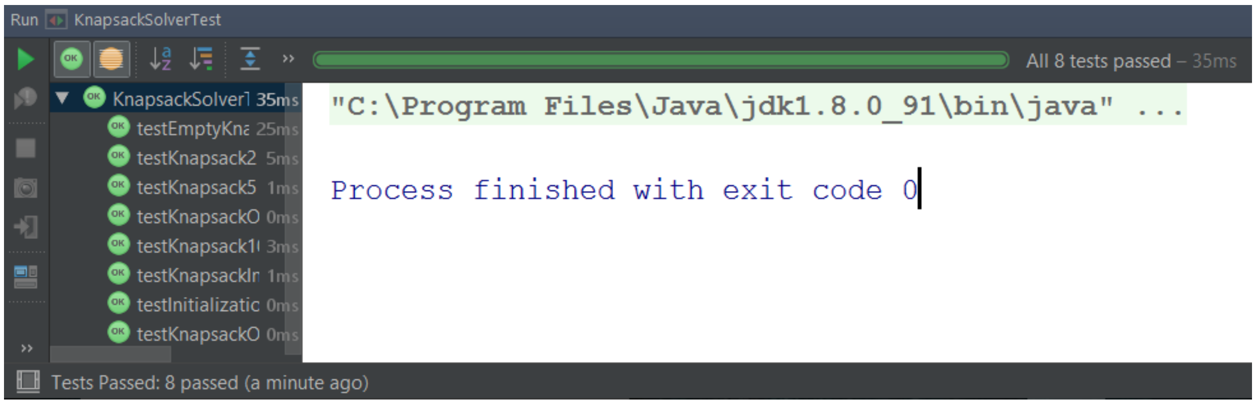
All unit tests passing (above).



Efficiency of N Queens (above).

Analysis of N Queens: As you can tell by the sheer number of points on this line graph, when solving N queens from 4-12, a wide variety of things happen. For the most part, as the number of queens increases, the amount of time taken to solve that board increases. However, you will notice that the time taken to solve 7 queens is the fastest out of all queens being solved for. This is simply because the number of recursive steps required to solve 7 queens is significantly less than that of 8 or 12 queens. At worst case, this program would need to go through every possible permutation of queens to obtain the correct solution, and due to that fact, I am stating that the complexity of this recursive program is O(N!). The time taken to solve the board does not purely rely on the number of queens to solve for, but how many times the program must backtrack to obtain the correct answer.

Knapsack Problem: The third, and final recursive program that I was responsible for creating was one that could solve the classic 1/0 knapsack problem. The concept is that with a given array of items, each item having a weight and a dollar value, what combination of items will get me the highest value without exceeding the capacity of my knapsack? I began my approach to solving this problem quite ambitiously, and eventually fell into a trap in which I was unable to properly backtrack. I spent several days in this confused state, until I had the pleasure of communicating with the person who designed this project, Richard Lester. We discussed the fundamental thought process that went into solving such a problem and I was soon able to come to the conclusion of what I had to do. The most basic thing I could do to approach the solution to this problem was to create two helper methods that would obtain the current weight of the knapsack based on the Boolean array that was passed into it, and comparing it to the item array, as well as a helper method to calculate the total value of said knapsack in the same way. The key to this problem lies within the non-recursive function that calls knapsackRecursive(). It is within this method that I created a Boolean array of length zero, and passed it into the recursive method. The base case for my recursive method is when the Boolean array passed into it (prior) is the same length as the item array. Now how this works is that with each all to the recursive method, I create two separate "working copy" arrays to mess with. These copies represent a divergent path in which I check for a solution. The first Boolean array represents the current Boolean array with the next item included, and the other array represents the current Boolean array with the next item excluded. Upon each recursive call, these two arrays would be constructed to be the length of the prior Boolean array plus one, which would allow me to eventually end up at my base case. I also created two arrays that would store the final solution to be returned to the user which I named "solutionWithNextItem" and "solutionWithoutNextItem" respectively. The last preparation step was to set up two double variables that would store the value of the solution knapsacks (I initialized them to -1.0, as this would allow me to determine if one was not initialized properly). To actually go about solving the problem all I needed to do was include some conditionals that checked to make sure that the arrays representing with and without the next item were not exceeding the capacity of the knapsack, set my solution arrays to the recursive call, calculate their value, compare them, and return whichever had the greatest value! Posted below are some screenshots depicting the unit tests passing, as well as a graph to analyze the efficiency of my KnapsackSolver algorithm.

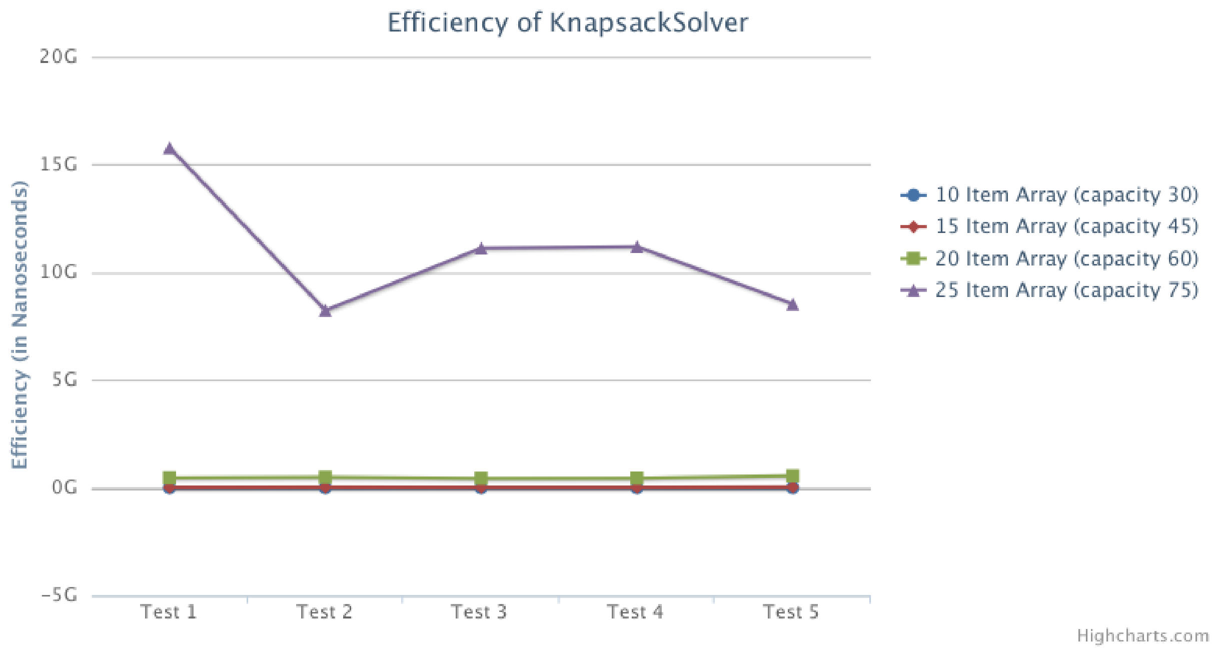


```
KnapsackSolverT 35ms
    testEmptyKna 25ms
    testKnapsack2 5ms
    testKnapsack5 1ms
    testKnapsackO 0ms
    testKnapsack1( 3ms
    testKnapsackIr 1ms
    testInitializatic 0ms
    testKnapsackO 0ms
```

```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...

Process finished with exit code 0
```

Tests Passed: 8 passed (a minute ago)

All unit tests passing (above).



Efficiency of Knapsack Solving Algorithm (above).

Analysis of Knapsack Solver: Although the above graph appears to only have three lines, I can assure you that there are indeed four. The 10 item array is simply masked by the fifteen item array, as the time taken to solve both was relatively close. What you will notice about the graph however, is that there is a tremendous leap in the amount of time taken to solve the 20 item array and the 25 item array. This has to deal not only with the amount of items in the array, but the capacity of the knapsack as well. For each item array, the capacity of the corresponding knapsack is three times the length of said array. By this, you can determine that a 25 item array, with weights and values between 0.00 and 9.00 has to calculate an incredible amount of permutations to return the best solution. With that being said, it is safe to say that at worst case, the program must calculate every single permutation possible to obtain the best orientation of the knapsack, leading to my conclusion that my knapsack algorithm is also O(N!).

Conclusion: How does one reflect on such a tremendous challenge that was laid in front of them? Well I can assure you, as I reflect on the many hours I have spent dedicating my life to this project, that I have definitely learned the absolute power of recursion. To get to the final product, took me approximately three weeks of pain, stress, and tears. That, in the end, however, was my biggest ally. I pride myself on my ability to manage my time and to get things done in the time that I am allotted. Not only did I struggle through this project and come out on top, I was also responsible for teaching several students in this class how the project worked as well. I have gained a deeper knowledge of how these programs function on a fundamental level, through verbally explaining the process to my peers. Unlike those who attempt to hurriedly complete this project on the day that it is due, my process has allowed me to fully tackle these problems to the best of my ability. In the long run, this project has made me a better programmer and pushed me beyond my limits.