Jasque Saydyk

Professor Patrick Kelley

CS 249 Section 3140

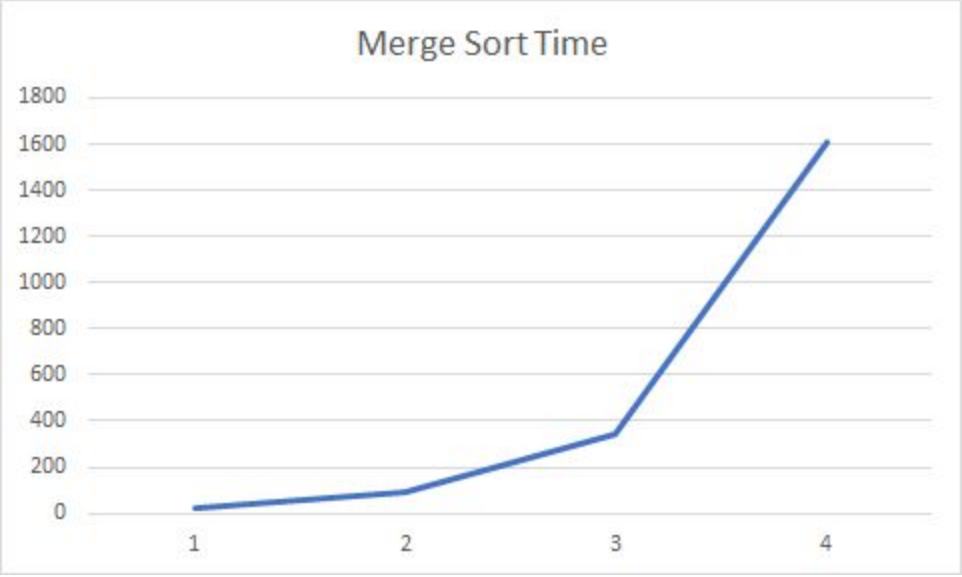
19 March 2017

Project 3 - Recursion

Merge Sort

For this part of the project, I decided to implement a merge sort by only manipulating the indices of the merge sort, rather than creating new arrays and merging them together. Doing this will save memory on the computer and accomplish the same task. In addition to this, I had to write a generic Comparator for the class to use to compare objects when merging the list back together. The biggest difficulty I had, besides having to research a ton about the merge sort to properly implement it, was getting the merge function to work correctly.

Below is a chart showing the time, in milliseconds, it took for the merge sort to sort lists of 20,000 items to 2,500,000 items. As we can see in the chart below, as the size gets larger so does the time it takes for the sort to complete, taking the form of an exponential looking curve. This makes sense as looking through my implementation, which makes liberal use of while loops, this would make it $O(n^2)$ as the recursion can be seen as a form of loop itself. This algorithm can certainly be improved upon, as the merge sort is capable of achieving better efficiency to my knowledge.

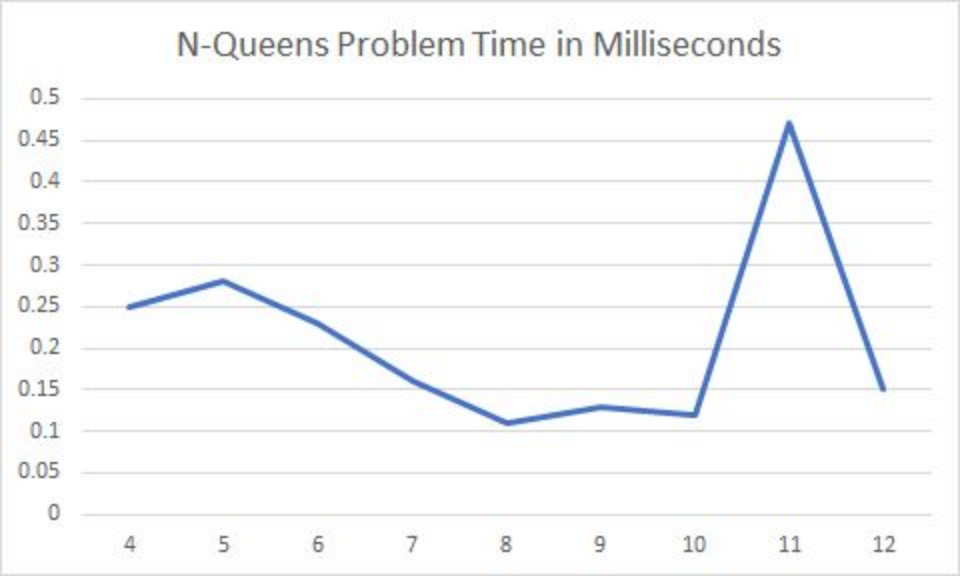


N-Queens Problem

For this problem, I attempted to solve it by creating a separate function that can also be passed in the current number of Queens on the board, then having it check to see if any queens are on the row and column, and if not, place the queen. If it does reach a point where it cannot place a queen, it will backtrack until it can, or output null.

I was able to complete most of this problem, and I feel like I am so close to the solution, however I cannot figure out the logic to check the solution to see if it is correct or not. As this is one of the most crucial parts of the program that actually gives you the correct answer or not, I was unable to correct complete this part of the project.

As for the algorithm analysis, the chart below shows the times for N values from 4 to 12 in milliseconds. While the below chart looks bumpy, the times differ only by half a millisecond, making it pretty flat and constant. The reason for this is because the N-Queens algorithm that I have doesn't actually find the solution, but quits rather quickly, thus causing these results. If my algorithm were to work correctly, I would expect it to be $O(n^2)$, which is due to the recursion, which acts like a look, and the series of the for loops in the check method that would cause this.



The 1/0 Knapsack Problem

For this problem, I attempted to solve it by creating another function that out also keep track of the current point in the list the recursion was on, then I would have it terminate when that variable would get larger than the list or the capacity got below zero. I would then skip any items that would break the bag with to much weight, then have the function recurse back on itself, once with the next item and another without. After this, my program would attempt to get the

value of the bag, and return the bag with the greater value. I failed at this last section, as I was unable to create the logic to do this, as my current logic fails to do it. After some discussions with the Instructor of the class, I was given the tip that I approached the problem wrong, and should instead look into building the lists in the recursion, rather than keeping track of the point in the list I was in. Despite this tip, I never recieved the time to implement it in that fashion.

Looking at the graph below, I ran the class I had with 10, 15, 20, and 25 items. We can see that the recursion aspect of the solution is definitely working, as the time raises with larger loads. Due to the sharp rise in the time spent on the larger problems, and after observing my code, it seems as though this algorithm has an $O(n^2)$ complexity, caused by the nesting of the recursion, which acts like a loop, and the for loop in the code.