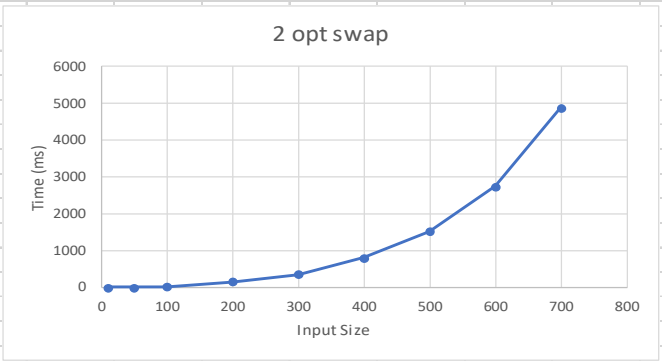
2optSwap Writeup

Algorithms and Data Structures

For my implementation of the 2optSwap I had runtimes that were surprisingly fast. I believe that my code works from what I understand about the algorithm, but I never had runtimes of over a minute. Even arrays of size 700 take around 5,000ms to complete, and an array of that size going through and checking if the array is correct by hand would take forever however, when run on smaller input sizes it seems to output the correct array to get the smallest route. I struggled with selecting the i and k values at first, having the user input them did not seem to work with how I was storing the routes. In searching online, I found come c++ implementations that seemed to have a helper function outside that did the cutting, reversing, and concatenating back together. So, I used this to have a second function that simply did the modifications and then called that function inside a nested for loop using the i and k from the for loops as the i and k in the 2optSwap. After getting the i and k values I push the modified array to a cache and store all possible permutations of routes based off i and k. I then pop arrays off cache and check their values and store the smallest array, once I have popped all arrays out of cache and checked their size, I return the array with the shortest route. From my runtime plot, it looks like the empirical runtime of my program is running in exponential time. Where the worst-case asymptotic time complexity would be $O(n^2)$. I get this from running my 2 nested for loops in the generating of the 2optSwap. The nested for loops iterate over the entire input size once for the outer loop and once for the inner loop. Within the inner loop I am performing a constant time dissection of pieces of the input array and storing these different arrays in another array making the worst-case time complexity for my program $O(n^2)$. The space complexity would also be $O(n^2)$, this is because an array takes up size $O(n)$ space and the only time that I have more space allocated than for one total array is when I call the 2optSwap function and push it to cache, I am creating the space for the input array to be iterated over as well as the space for the array to be altered and pushed into cache. After I have both cache and the input array, I am decrementing cache and storing those values in other variables set to be no greater than cache itself. So, the worst case space complexity for my program would be that I have 2 full arrays of size n or $O(n^2)$.

Input Size	Time (ms)
10	0.707
50	6.03
100	18.259
200	143.94
300	359.197
400	797.775
500	1523.288
600	2739.731
700	4878.501

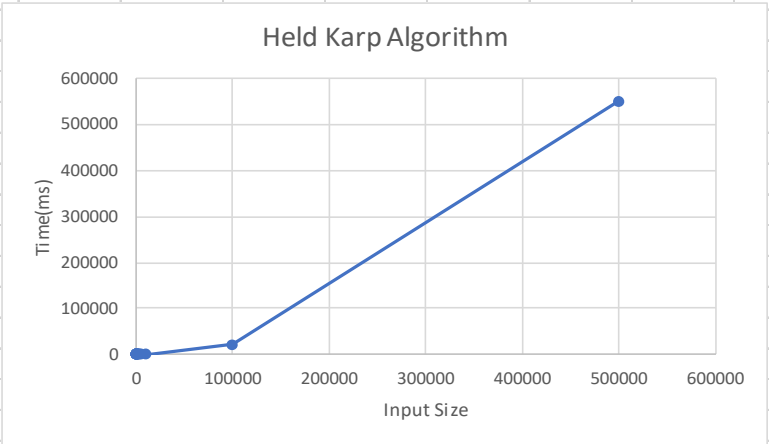


Held Karp Writeup

Algorithms and Data Structures

The runtimes of my implementation were at times too long to compute, I did not have enough time to run everything on AWS even after hours elapsed. This leads me to believe that the empirical runtime is much higher than the graph leads me to believe. From the inputs the times go from mere milliseconds to minutes after 100,000 input cities. If I ran the program on inputs over 500,000 it seemed like the program would never end. The largest input would have had a runtime of at least an hour was a very true statement. At 500,000 input cities the runtime was about 9 minutes. The worst case asymptotic time complexity of my algorithm would be $O(2^n n^2)$, this is because within the nested for loop I am iterating over arrays of n length, checking the distance of the arrays I have iterated over, and then checking the distance against the initial starting distance, and finally depending on which array has the smallest total distance I am storing that total distance and checking it against all the other distances of the arrays. Whichever one has the smallest total distance in the end is the route and then I return that distance. Space wise though, the complexity of my algorithm as a worst case would have a slightly smaller impact but ultimately still quite large with $O(2^n n)$, as a worst case having to store multiple different arrays because the total distance would cause the space to get this high. Where I am only testing and then throwing the larger values out, I don't think my program gets quite that close in terms of space usage, but with the absurdly large runtimes it could after a certain input size.

Input Size	Time(ms)
10	0.804
50	0.133
100	0.482
200	1.819
300	0.214
400	0.36
500	0.674
600	0.775
700	1.009
800	1.333
900	1.764
1000	2.06
2000	9.039
3000	19.842
5000	56.025
10000	215.623
100000	22012.55
500000	551233.571



Comparison

Algorithms and Data Structures

The empirical time of both of my implementations show signs of being similar in the beginning but towards then end, the held karp algorithm runs away with the total time taken this could be because my held karp can manage large input sizes where my 2opt could not. My held karp implementation was able to handle much larger input sizes than my 2 opt swap, but if I could run that large of inputs on my 2 opt swap the data leads me to believe that I would find the best route faster using 2opt over held karp. Looking at the plot points though for each same input and same test environment my held karp implementation returned results in a much faster time than my 2opt. So, my held karp can handle larger input sizes and compute smaller ones in a faster time, this seems to go against what I know should be happening based off what I know about the algorithms. But I am unsure where this differentiation comes from. Empirically 2opt looks better and smoother and more precise, but overall held karp has the better results.