1. **The Algorithm**

We are currently running multiple "final algorithms" in order to get overall optimal outputs. Our main algorithm is a version of dynamic programming, where we start by finding all feasible single-attraction sequences. We initialize a list of valid paths to these sequences, and a list of all paths to these sequences. We then iterate through each valid path, checking which single-attraction sequences can be added without making the path infeasible. All paths which can be extended are extended and stored in valid paths. These valid paths are appended to all paths, and then we reiterate through all the new valid paths, checking which single-attraction sequences can be added. This algorithm obviously takes far too long.

To shrink runtime, we also introduced the concept of "rank" which keeps a cutoff for the number of valid paths that can be fed into each iteration of the algorithm. For example, if rank=500, then, each time after the new list of valids is returned from one longer, we sort the paths by some attribute (found by a capping_function) and set valid paths to only the top 500 sequences. Then, the next time we run one longer, it only runs it on a maximum of 500 paths. At the very end of our algorithm we go through all the feasible paths and find the one with the highest utility.

We varied this algorithm up by changing how we did our rank. We ranked them by utility, time (keep the shortest paths), efficiency (keep the highest efficiency paths), and differently weighted combinations of multiple of them. The rank cutoff was constantly varied. We had the most faith in efficiency as a capping function, so it ran with ranks 1,000,000 for smalls, 100,000 for mediums, and 600 for larges, but other functions were run with varied ranks to keep runtime shorter.

Our second algorithm was a simple greedy algorithm. The algorithm works by starting at (200,200) and scanning all of the attractions which are feasible to go to (can go there, use it, and get home before closing), and chooses the algorithm with some highest quality. Then, the algorithm adds that attraction to the path, updates location variables to that attraction, and scans again for the next feasible attraction with highest efficiency. We varied this algorithm by changing the qualities to highest utility-time ratio, highest utility, lowest total path time (also including one with time to get home), earliest closing time, earliest opening time, closest distance, smallest availability (closing time + opening time), shortest duration attraction, and more.

We also varied the greedy algorithms by finding the highest average utility possible from choosing an available attraction and the next available attraction. For example, when checking the next attraction to go to in terms of utility, it would also greedily check what the next attraction after that would be from the end of the first attraction. Then, it would average the utility of both of those attractions, and go to the attraction with the highest average between it and the following attraction. We also tried doing this a third time (check what the next 3 attractions are for each attraction visited), but that took far too long for a quick checking algorithm, the dynamic solution was the time-heavy one and this was more to cover some blind spots.

Overall, we are very happy with how our algorithms performed and believe that the many variations we used of our algorithms did a good job at covering the others' "blind spots." Our performance was pretty good, and looking at the leaderboard, we got the optimal solution or close to the optimal solution on most of the inputs. That being said, there were also some ideas which we had to abandon due to both time constraints and utility. One of them was a joined path algorithm, which worked like our dynamic programming one, but instead of lengthening paths attraction by attraction, it would have joined valid paths together to create paths of length 2n-1. We abandoned this because it would have left many holes in the paths, and even if it did run faster, it would involve a lot more post-processing in order to figure out which groups to put together. Since our attraction-by-attraction algorithm worked well, we felt this wasn't worth our time. We also thought of making an algorithm to take out duplicate attractions, so it would view them as one and use them for the count that they have. For example, if attraction a = attraction b, then increasing the path by a gives essentially the same path and we don't need to do it twice. However, we also abandoned this because there were not many inputs with completely identical attractions, and this algorithm would have to do much rounding to have any utility.

2. **Running Our Code**

In all honesty, we should've read this before we ran our algorithms so we could keep track of all the tests we ran. We copied the jupyter notebook and just ran tons of ranks and weights and capping functions over the course of about a week on each of our computers. The jupyter notebook, TheoryWorld.ipynb, explains what most functions do, but the function *run* in the dynamic programming section runs the main dynamic programming solution with given ranks and capping functions etc. The maximum rank we used for smalls was 1,000,000; for mediums was 100,000; and for larges was 600 with a capping function set to maximize efficiency.

The last cell runs all greedy algorithms and takes the maximum output for each park.

We manually solved thecoolguys_medium3 early on in our process. We were looking at the ones we missed with the schedule visualizer and the location visualizer and realized it would be easy to solve by hand so we did.

We didn't continue solving parks after we had the best score on the leaderboard. This is further explained in 3.

3. **External Resource Usage**

We used google colab in the beginning so everyone had access to our code. It runs slower than all our computers though and uploading files to it is a pain so we downloaded the notebook afterwards and just ran locally, merging outputs frequently.

We used RStudio and R to create a schedule visualizer, which was posted to Ed during the input phase. We used it for our manual input and to find what kinds of inputs our algorithm struggled with.

We used an excel document to track how we, and others, were performing on the leaderboard. This was vital to keeping our runtime down. We would've written a web-scraping script to keep constant track of the maximums, but it was simpler to just copy-paste the leaderboard every few days. With the document keeping track of which inputs we had the top score of, we were able to take these inputs out of the pool of algorithms to test on. It might've meant we missed being the only ones to maximize a solution here or there, but it cut our runtime enormously. For the last few days, we were only running on 55-65 inputs total instead of 360.

4. Team Reflection

We met together at least once a week since the start of the project and almost every day for the last week to discuss observations, merge outcomes, and brainstorm alterations to the algorithms. Everyone contributed code (which made merging annoying sometimes, but we figured it out). Having time to talk through observations was super helpful because we could all take advantage of each other's successes, but it was also vital to work on our own and find creative solutions that others might not have seen.

The best thing we did as a team was create infrastructure in the input phase. By the time we turned in our inputs, we had, in python, an attraction class fully defined with setter functions that could read inputs or generate random lists of attractions. It could also take lists of attractions and find their ids (to later be translated into outputs), check if paths were valid, check where you ended up after an attraction finished. This infrastructure made working on algorithms and debugging way easier than if we were constantly indexing through raw tuples. Additionally, we came up with a bunch of algorithms in the beginning and settled on one to follow two weeks out. This let us have a week for development of it and a week to just run and modify as we saw fit.

As we worked on the project we were surprised by the amount of optimal or near-optimal outputs we could obtain by just running simple greedy algorithms. Sometimes, reversing the capping function to find "worst case" algorithms also ended up finding some optimal outputs. Or weighting utility and time almost randomly to get paths we might not have seen before.

We would've pursued a much stronger set of greedy algorithms (and a more generalizable one so we weren't copy and pasting. This could've filled holes in our dynamic solution, which struggled to find optimal solutions to algorithms with many available attractions and a long optimal sequence. We probably also would've implemented a better version-control system, constantly pushing outputs to our own projects and then keeping a master list that only updated after we had all pushed, taking the maximum of our outputs.

If we could change the project itself, we would make teams not allowed to include special symbols or emojis in their team names as this caused extra problems and work that wasn't related to the project itself. It also could be useful to talk about the project during a discussion to give everyone an idea of what was happening instead of relying on Ed for communication. More staff inputs also could be useful as it makes it easier to distinguish between team scores.
All-in-all it was a great project and we enjoyed it, but on top of extensive weekly problem sets and two tests, it was fairly robust. Competition is fun and great, but also pushes us to put in as

much work as possible, ensuring certain grades so that we know when we could stop working might help. The biggest problems were because it was the first iteration of the project (project spec was confusing often, special character compression issues, getting TheoryWorld up and running), but they were addressed so quickly so thank you! Also, can TheoryWorld allow like, 3 or 5 requests per second instead of 1? Every time we hit the back button or misclicked it freaked out.