Readme_aarment2

1. **Team name:** aarment2
2. **Names of all team members:** Alan Armenta
3. **Link to github repository:** https://github.com/ArmeAE22/aarment2_theory_proj1.git
4. **Which project options were attempted:** Rewrite DumbSAT to use an incremental search through possible solutions
5. **Approximately total time spent on project:** 8-10 Hours
6. **The language you used, and a list of libraries you invoked:** Python, matplotlib
7. **How would a TA run your program (did you provide a script to run a test case?):** Yes, if you run the program without an input it will run one of the given test cases in the DumbSAT template. You can change this default in line 300 (TestCases = X). I provided three input files with small, medium, and large problem size test cases. You just need to type in the file name as a command line argument. For the plots, the program will automatically update the plots_aarment2.png file which shows the problem size on the x-axis and time on the y-axis. For the check input files, if you run the medium or large test input files it will automatically update the respective plot file.
8. **A brief description of the key data structures you used, and how the program functioned.** To implement an incremental search, I used a recursive DFS function instead of the brute force loop used in the dumbSAT example. The program runs very similarly to the dumbSAT program since I used it as a template. The main change is in the check() function which is now incremental rather than brute force.
9. **A discussion as to what test cases you added and why you decided to add them (what did they tell you about the correctness of your code). Where did the data come from? (course website, handcrafted, a data generator, other)**
   To verify the correctness of my code, I kept the dumbSAT program and ran the same test cases on both programs. The dumbSAT program uses a brute force search approach, which takes a long time but it's guaranteed to find a solution if there exists one. I used this to my advantage and ran two terminals to compare the results. This was also helpful when approximating the time complexity, since they both output the execution times. I used the randomized test cases generated inside the DumbSAT template to see trends in the complexity. The program generates a very good variety of random test cases, reaching a very high number of variables and clauses. I used them because I could quickly see averages for different amounts of variables/clauses. All I had to do was run the program and it would generate new test cases for me, ranging from 2 variables all the way up to 100 or more if I waited long enough. I was getting consistent results within given ranges of variables/clauses, which tells me the incremental search is consistent and correct.
10. **An analysis of the results, such as if timings were called for, which plots showed what? What was the approximate complexity of your program?** Since I used a recursive DFS increment search, the execution times were much lower than the DumbSAT brute force search method. The plots_aarment2.png file automatically updates every time you run the program and it shows the problem size on the x-axis and time on the y-axis. The plots_mediumcases_aarment2.png and plots_largecases_aarment2.png specifically show the plots for the check_medium and

check_large input files. I chose to plot these two because they show how the execution time climbs exponentially with problem size. At around 40, it starts to significantly spike which is confirmed by the medium cases plot. Based on the graphs, it's hard to exactly pinpoint the time complexity of this program, but on average the time complexity of a recursive program is 2^n. The function calls itself twice to check true and false assignments. However, one thing to keep in mind with the incremental search used here was that the execution times were heavily tied to the number of clauses. This makes sense since the program has to check all clauses with the current truth value assignments to the variables. With this in mind, the time complexity of this program is O(2^n * clauses). To make sure my program was successfully finding solutions, I plugged in the same test cases to both the brute force template and my DFS program. I made sure that if the brute force template returned a satisfiable solution, my program also did the same.

11. **A description of how you managed the code development and testing.** I took time to go over the DumbSAT example template to see how I would go about rewriting it to use an increment search. I started with just using the test_wff() and check() functions to test if it correctly generated a solution to wffs of various sizes. I plugged in a wff which I knew was satisfiable into the DumbSAT template and checked if I also got a solution in my own incremental version. I started small with 2-3 clauses and variables, then worked my way up. I kept going until I was confident that the incremental search program was working correctly. Eventually, I implemented the build_wff and run_cases functions from the DumbSAT template to use the randomized test cases. These worked well to show the trends and since they were the same functions I was able to compare my iterative search times with the brute force times. The program generates a very good variety of random test cases with it reaching a very high number of variables and clauses.
12. **Did you do any extra programs, or attempted any extra test cases.** The DumbSAT template was the only extra program I used to validate my results.