Project Readme x Team Logan:)

Version 1 9/11/24

A single copy of this template should be filled out and submitted with each project submission, regardless of the number of students on the team. It should have the name readme_"teamname"

Also change the title of this template to "Project x Readme Team xxx"

<table>
<tr><td>1</td><td>Team Name: Logan:)</td></tr>
<tr><td>2</td><td>Team members names and netids: Ristian David mdavid4</td></tr>
<tr><td>3</td><td>Overall project attempted, with sub-projects: SAT: Brute force & Best Case</td></tr>
<tr><td>4</td><td>Overall success of the project: Successful</td></tr>
<tr><td>5</td><td>Approximately total time (in hours) to complete: 6 hours</td></tr>
<tr><td>6</td><td>Link to github repository: https://github.com/LoganStPierre/Project1-TOC</td></tr>
<tr><td>7</td><td>List of included files (if you have many files of a certain type, such as test files of different sizes, list just the folder): (Add more rows as necessary). Add more rows as necessary.
<table>
<tr><td>File/folder Name</td><td>File Contents and Use</td></tr>
<tr><td colspan="2">Code Files</td></tr>
<tr><td><code>src/helpers/sat_solver_helper.py</code></td><td>Modified to match output file name scheme</td></tr>
<tr><td><code>src/helpers/constants.py</code></td><td>Modified to have correct input file selection</td></tr>
<tr><td><code>src/sat.py</code></td><td>Implemented different algorithms, brute force and best case</td></tr>
<tr><td><code>configuration/student_config.json</code></td><td>Configuration for the uv to run the proper algorithms</td></tr>
<tr><td colspan="2">Test Files</td></tr>
<tr><td><code>2SAT.cnf</code></td><td>Input test file. Has sat up to 20 variables. Much more is very slow and not feasible for bruteforcing I have found.</td></tr>
<tr><td colspan="2">Output Files</td></tr>
<tr><td>output_brute_force_2SAT</td><td>The file format is</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td>_sat_solver_results_Logan_:).csv</td><td>output_{subproblem}_{inputfile}_sat_solver_results_Logan_:).csv the contents are as expected in the provided results writer</td></tr>
<tr><td colspan="2">Plots (as needed)</td></tr>
<tr><td>plot_brute_force_Logan_\:\).png</td><td>This is an image of the bruteforce timing data graphed in desmos. I parched the data from the output file with this command: cat output_brute_force_2SAT_sat_solver_results_Logan_:).csv |awk -F, 'NR > 1 { print $2 "," $6 }' then i added ()’s and graphed it on desmos after some clean up</td></tr>
<tr><td>plot_best_case_Logan_\:\).png</td><td>This plot was created using matplotlib and the output generated from the 2sat.cnf file ran through the best case solver</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>Programming languages used, and associated libraries: Python, Matplotlib.
Brute force timing data was collected through an awk of the output file and graphed on Desmos.
Matplotlib was used for graphing of the best-case timing data.</td></tr>
<tr><td>9</td><td>Key data structures (for each sub-project):
Brute force:
● looped over arrays of dictionaries in order to individually modify them and check posed solution
Best Case:
● Simple loop over dictionaries, lists of lists and an integer bitmask loop to generate all possible assignments.</td></tr>
<tr><td>10</td><td>General operation of code (for each subproject):
Brute force:
● The brute forcing algorithm reads in the problems line by line and attempts all possible sets of values through iteration. These iterations are then checked as created to see if they are acceptable. If they are the machine stops and returns the correct values.
Best Case:
● I generate every possible T/F assignment using bitmasks and for each i check every clause to see how many are satisfied. If one assignment satisfies all clauses, it returns immediately. I nothing satisfies all, then the one with the most is returned.</td></tr>
<tr><td>11</td><td>What test cases you used/added, why you used them, what did they tell you about the correctness of your code.
Brute force:
● I used the provided 2sat.cnf, which I modified to use spaces rather than commas per the program spec. This file contains 2SAT problems with up to 20 variables. This test input file revealed to me what was later confirmed by the plotting: this algorithm is very slow at higher variable numbers. I eventually reduced the size</td></tr>
</table>

	of the test file, and the processing speed became much more manageable. This code is correct as tested, but it uses a slow methodology. Best Case: I uses the 2SAT.cnf which I also modified to use spaces rather than commas. I made some smaller test cases when building to check as I went along. These tests showed that the code worked properly and my logic behaved as I thought.
12	How you managed the code development: used github operations. Ensured individual tasks understood and timely completed.
13	Detailed discussion of results: Brute force: ● As discussed in test cases, the brute force algorithm is very slow at harder problems. The graph shows a clear exponential curve associating the time required with the number of variables. This is expected with a lazy algorithm. The code was fairly straightforward to write, meaning slowdowns from python operations is fairly limited, so with problems of reasonable size, the program finishes quite quickly and immediately with tautologies. Best Case: ● It worked as expected. The SAT problems usually finished fast because the solver stops as soon as it finds a working assignment, The Unsat took much longer since every possible assignment had to be checked and the timing graph shows this clearly with the unsat points rising much faster.
14	How team was organized: The team analysised the instructions then worked out how we ought to approach it. We worked on our individual function assignments separately and regathered to factor everything together as needed. We communicated via text as needed while working individually. This was most useful in preventing the other from making similar mistakes or quickening our understanding of the structure of the given code skeleton.
15	What you might do differently if you did the project again: I would make it more collaborative. I think we learn much better when we sit together and design functions on a whiteboard and then code them out, or at least that is how I (Ristian) work best. When we do it next time we might do this exercise to ensure that both of us learn what each other coded at a deeper level not just a high level overview.
16	Any additional material: