Problem assignment 2: Solving MAXSAT instances.

Ali Ahammad (a81317)

Raul Lopes (a86079)

Course: Metaheuristics

Objective:

In computational complexity theory, the maximum satisfiability problem (MAX-SAT) is the problem of determining the maximum number of clauses, of a given Boolean formula in conjunctive normal form, which can be made true by an assignment of truth values to the variables of the formula. It is a generalization of the Boolean satisfiability problem, which asks whether there exists a truth assignment that makes all clauses true. Hill climbing is mathematical optimization technique which belongs to the family of local search.

Three DIMACS instances have been given:

- uf20-01.cnf : 20 variables, 91 clauses
- uf100-01.cnf : 100 variables, 430 clauses
- uf250-01.cnf : 250 variables, 1065 clauses

We have to implement the following algorithms under certain conditions:

- Next Ascent Hill climbing using 1-bit Hamming Distance neighbourhood.
- Multi start Next Ascent Hill climbing (MSNAHC)
- Variable Neighbourhood Ascent using up to a 3-bit Hamming Distance neighbourhood.
- Multi start Variable Neighbourhood Ascent (MSVNA)

Next Ascent Hill climbing using 1-bit Hamming Distance neighbourhood:

Tasks:

1) The neighbourhood of the current solution should always be visited in a random order, and not from left to right, or some other pre-defined order.
2) Test the algorithm with three DIMACS instances.
3) Execute 30 independent runs the algorithm for each problem instance, and report the results obtained.
4) For each instance report the best solution quality obtained, the number of objective function evaluations needed to reach that solution, and the CPU time taken to do so.

Algorithm and Pseudocode:

Random Exploration: The primary objective was to implement Next Ascent Hill Climbing for solving MAXSAT with a 1-bit Hamming distance neighbourhood. The key modification was to visit the neighbourhood in a random order, ensuring the algorithm does not follow a fixed left-to-right strategy. This randomized exploration increases the chances of escaping local optima early in the search, enhancing its performance compared to a deterministic approach. The neighbourhood of the current solution is always visited in a random order using *random.shuffle(neighbours)*.

DIMACS Instances: The algorithm is tested with three DIMACS instances: uf20-01.cnf, uf100-01.cnf, and uf250-01.cnf.

Execution: The algorithm is executed 30 times independently for each problem instance. The results, including the best solution quality, the number of objective function evaluations, and the CPU time taken, are reported for each instance.

Here is the pseudocode:

```
 1.  Input: MAXSAT problem instance, number of variables (num_vars), number of clauses
(num_clauses), max_iterations, num_runs
 2. Output: Best solution found, number of objective function evaluations, CPU time
 3.
 4.  Initialize problem instance from DIMACS file.
 5.    a. Read num_vars and num_clauses from the problem file
 6.    b. Parse the clauses
 7.
 8.  Function EvaluateFitness(solution)
 9.    a. satisfied_clauses = 0
10.    b. For each clause in clauses:
11.       i. If the clause is satisfied by the current solution, increment satisfied_clauses
12.    c. Return satisfied_clauses
13.
14. Function GenerateNeighbours(solution)
15.    a. neighbours = []
16.    b. For each bit in solution:
17.       i. Flip the bit to generate a neighbour
18.       ii. Append the neighbour to neighbours.
19.    c. Return neighbours
20.
21.  Function NextAscentHillClimbing(initial_solution, max_iterations)
22.    a. current_solution = initial_solution
23.    b. best_solution = current_solution
24.    c. best_fitness = EvaluateFitness(current_solution)
25.    d. For each iteration in max_iterations:
26.       i. Generate neighbours of current_solution
27.       ii. Shuffle the neighbours randomly.
28.       iii. For each neighbour:
29.          - Evaluate its fitness
30.          - If it improves the best_fitness:
31.              - Update best_solution and best_fitness
32.       iv. current_solution = best_solution
33.    e. Return best_solution
34.
35.  Main Algorithm:
36.    a. For each run in num_runs:
37.       i. Generate a random initial_solution
38.       ii. Start the timer.
39.       iii. Call NextAscentHillClimbing(initial_solution, max_iterations)
40.       iv. Record the time, best_solution, and best_fitness
41.       v. Print the result for this run
42.
43.  After 30 runs:
44.    a. Compute and print summary statistics:
45.       i. Average time
46.       ii. Maximum clauses satisfied.
47.       iii. Average clauses satisfied.
48.
```

Per□ormance Analysis and Results:

For the instance uf20-01.cnf:

- The maximum number of clauses satisfied in any run was 91/91.
- The average number of satisfied clauses across 30 runs was approximately 88.77.
- The average execution time per run was 11.23 seconds.

However, for the larger instances (uf100-01.cnf and uf250-01.cnf), the algorithm took considerably longer, making it impractical to run within a reasonable time frame on my current machine.

Challenges and reasoning:

The performance bottleneck for larger instances can be explained through the following factors:

- **Neighbourhood Size Growth:** As the number of variables increases, the size of the neighbourhood grows linearly. For a problem with n variables, the algorithm generates n neighbours at each iteration. In the uf250-01.cnf case, this means generating 250 neighbours per iteration, which directly increases the computational overhead for evaluating the fitness of each neighbour.
- **Fitness Evaluation Complexity:** The time complexity of evaluating each solution is proportional to the number of clauses in the problem. As the instance size increases, the number of clauses also increases significantly. For example:
    - uf20-01.cnf has 91 clauses.
    - uf100-01.cnf has 430 clauses.
    - uf250-01.cnf has 1065 clauses.

Evaluating the fitness for each neighbour involves checking all clauses, making the computational effort grow linearly with the number of clauses. For the larger instances, the time required for evaluating each neighbour becomes a major bottleneck.

- **Number of Objective Function Evaluations:** In metaheuristics, the number of objective function evaluations plays a critical role in determining overall performance. Since Next Ascent Hill Climbing explores every neighbour until it finds an improvement, the number of evaluations grows with the problem size, leading to longer execution times for larger instances.

Big O Complexity:

The Big O complexity for the Next Ascent Hill Climbing algorithm in the context of solving MAXSAT can be calculated as follows:

Neighbourhood Generation:

- For each solution of size n (number of variables), the algorithm generates n neighbours (1-bit Hamming distance neighbours).
- Time complexity for generating neighbours: **O(n)**

Fitness Evaluation:

- For each neighbour, the fitness function evaluates how many clauses are satisfied. This involves checking every clause (total m clauses in the problem).
- Time complexity for each fitness evaluation: **O(m)**

- Since there are n neighbours to evaluate, the total time for evaluating all neighbours is: **O(n * m)**

Shuffling Neighbours:

- Randomly shuffling the neighbours has a time complexity of **O(n)**.

Iterations:

- The algorithm performs the above steps for a maximum of k iterations (where k is the number of iterations allowed).
- Total time for all iterations: **O(k * n * m)**

Independent Runs:

- The algorithm runs independently r times (in my case, r = 30).
- Total time complexity: **O(r * k * n * m)**

Final Big O Complexity: Combining these, the overall time complexity is: **O(r * k * n * m),** Where:

- r: Number of independent runs (30 in my case).
- k: Maximum number of iterations (e.g., 10,000).
- n: Number of variables in the MAXSAT instance.
- m: Number of clauses in the MAXSAT instance.

In case of the instance *u□100-01.cn□*show the following:

- **Run 1**: Best solution satisfies 417/430 clauses, Time taken: 306.85 seconds
- **Run 2**: Best solution satisfies 416/430 clauses, Time taken: 385.49 seconds
- **Run 3**: Best solution satisfies 422/430 clauses, Time taken: 322.03 seconds
- **Run 4**: Best solution satisfies 425/430 clauses, Time taken: 323.38 seconds

Given the complexity calculation of **O(r * k * n * m)** for the algorithm, let's break down why it takes significantly longer compared to smaller instances.

- **Variables (n)**: In uf100-01.cnf, there are 100 variables. The algorithm generates n = 100 neighbours at each iteration.
- **Clauses (m)**: There are 430 clauses, and each neighbour's fitness evaluation requires checking all clauses to count how many are satisfied.
- **Iterations (k)**: The number of iterations (up to 10,000 in my case) further amplifies the time required for each independent run.

Thus, the time taken is proportional to the product of these three factors. For example, evaluating 100 neighbours per iteration and checking 430 clauses for each fitness evaluation means that a single iteration involves **O(n * m) = O(100 * 430) = 43,000** operations.

For 10,000 iterations, this results in **O(k * n * m) = O(10,000 * 100 * 430) = 430,000,000** operations, explaining the significant time for each run. Additionally, since I'm running the algorithm 30 times

(independent runs), the total time will scale linearly with r, increasing the overall execution time substantially.

After considering the above circumstances, the algorithm performs well in terms of solution quality, but the computational cost is high, particularly for larger instances.

Multi start Next Ascent Hill climbing (MSNAHC)

Tasks:

1) Repeat the previous question using Multi start Next Ascent Hill climbing (MSNAHC) instead of Next Ascent Hill climbing.
2) Again, perform 30 independent runs of my algorithm for each problem instance. Each execution of MSNAHC should stop if either a global optimum is found or if by the time a given restart begins a maximum of 10 million objective function evaluations have already elapsed, whichever occurs first.
3) For each instance report the best solution quality obtained, the number of objective function evaluations needed to reach that solution, and the CPU time taken to do so.

Algorithm and Pseudocode:

Random Exploration: The neighbourhood of the current solution is explored in a random order. This ensures that the algorithm is not biased towards specific bit positions in the solution vector, avoiding local optima caused by predetermined visitation patterns and the neighbourhood of the current solution is always visited in a random order. For each candidate solution, all neighbouring solutions (obtained by flipping a single bit in the current solution) are generated. These neighbours are randomly shuffled, and then the algorithm checks their fitness. If a better solution is found, it moves to that solution, and the process continues.

DIMACS instances: It evaluates the algorithm with three DIMACS instances (uf20-01.cnf, uf100-01.cnf, uf250-01.cnf).

Multiple Runs Execution: It executes 30 independent runs of the algorithm for each problem instance. For each instance, report the best solution quality obtained, the number of objective function evaluations needed to reach that solution, and the CPU time taken.

Here is the pseudocode:

```
1. Input: MAXSAT instance (clauses, number of variables), max_evaluations
2.  Initialize global_evaluations to 0
3.  Set best_solution to None, best_fitness to -1, and best_eval_count to 0
4.
5.  While global_evaluations < max_evaluations do:
6.     a. Generate a random initial_solution of 0s and 1s
7.     b. Evaluate current_fitness of initial_solution
8.     c. Set eval_count to 1
9.
10.    d. Repeat until no better neighbour is found or max_evaluations exceeded:
11.         i. Generate all 1-bit Hamming distance neighbours of current_solution
12.         ii. Shuffle neighbours to ensure random visitation order.
13.         iii. For each neighbour:
14.             - Evaluate its fitness
15.             - If neighbour's fitness is better than current_fitness:
```

```
                - Update current_solution and current_fitness
                - Break (go to next iteration)
            - Increment eval_count

    e. Increment global_evaluations by eval_count

    f. If current_fitness > best_fitness:
        i. Update best_solution, best_fitness, and best_eval_count

    g. If best_fitness equals number of clauses, break (solution found)

Return best_solution, best_fitness, best_eval_count, and global_evaluations
```

Performance Analysis and Results:

The algorithm was tested on the following MAXSAT instances:

- **uf20-01.cnf**: Contains 91 clauses and 20 variables.
- **uf100-01.cnf**: Contains 430 clauses and 100 variables.
- **uf250-01.cnf**: Contains 1065 clauses and 250 variables.

Results for each instance were recorded over 30 independent runs to assess the algorithm's performance in terms of solution quality, objective function evaluations, and CPU time.

For each of the three problem instances, the algorithm was executed 30 times. The following metrics were reported:

- **Best solution quality:** All 91 clauses were satisfied in several runs.
- **Average time:** Approximately 0.0604 seconds per run.
- **Average evaluations:** 762 evaluations per run.
- **Maximum clauses satisfied:** 91 (all clauses).

The algorithm performed very efficiently for this small instance.

Difficulty with Larger Instances (uf100-01.cnf, uf250-01.cnf)

For larger instances, such as uf250-01.cnf, the algorithm experienced prolonged execution times. This behaviour can be explained both mathematically and logically:

- **Exponential Growth in Neighbourhood Size**: The neighbourhood size grows linearly with the number of variables. With 250 variables, each solution has 250 neighbouring solutions. This dramatically increases the time spent evaluating neighbours.
- **Objective Function Evaluations**: Evaluating the fitness of each neighbouring solution involves checking each clause in the SAT instance. As the number of clauses increases, so does the computational effort per evaluation.
- **Search Space**: The search space grows exponentially with the number of variables (2^n). Although MSNAHC explores a small portion of this space, larger instances increase the number of local optima, making it harder to find global optima efficiently.

Big-O Complexity Analysis

The worst-case complexity for the algorithm can be estimated as:

- Generating and evaluating neighbours for each solution involves O(n) operations, where n is the number of variables.
- The total number of evaluations can grow linearly with the number of variables and clauses, O(m * n), where m is the number of clauses.
- Given that the algorithm stops after a fixed number of evaluations, the overall complexity is O(max_evaluations * n).

The MSNAHC algorithm performed well for small instances such as *uf20-01.cnf*. However, the computation time increased dramatically for larger instances like *uf100-01.cnf* and *uf250-01.cnf*, due to the growing search space and neighbourhood size.

Variable Neighbourhood Ascent using up to a 3-bit Hamming Distance neighbourhood.

Tasks:

1. Test it on the 3 problem instances mentioned above.
2. Again, perform 30 independent runs of my algorithm for each problem instance,
3. Report the results obtained. For each instance report the best solution quality obtained, the number of objective function evaluations needed to reach that solution, the CPU time taken to do so.

Algorithm and Pseudocode:

Neighbourhood Order: The VNA algorithm iteratively explores solutions in neighbourhoods defined by flipping 1, 2, or 3 bits of the current solution (k-bit Hamming Distance neighbourhoods). If no improvement is found in a neighbourhood, the algorithm increases the neighbourhood size (i.e., k) until it reaches a predefined limit. The neighbourhood is shuffled randomly to ensure diverse search directions. The neighbourhood of the current solution is visited in a random order using *random.shuffle(neighbours)*.

DIMACS instances: It evaluates the algorithm with three DIMACS instances (uf20-01.cnf, uf100-01.cnf, uf250-01.cnf).

Multiple Runs Execution: Each instance is tested for 30 independent runs. The algorithm seeks to maximize the number of satisfied clauses by iteratively searching through different neighbourhoods of the current solution.

Here is the pseudocode:

```
 1. Input: Problem instance (clauses from a DIMACS CNF file), number of variables (num_vars),
number of runs (num_runs)
 2.
 3.  Load the DIMACS CNF file and parse the clauses.
 4.
 5.  Define evaluate_fitness(solution, clauses):
 6.      - For each clause, check if any literal in the clause is satisfied.
```

```
    - Return the number of satisfied clauses.

Define generate_k_bit_neighbours(solution, k):
    - Generate all possible neighbours of the current solution by flipping k bits.

Define variable_neighbourhood_ascent(initial_solution, clauses, max_iterations):
    - Initialize current_solution with initial_solution.
    - Set best_solution to current_solution and best_fitness to its fitness value.
    - Set k to 1 (neighbourhood size).

    While k ≤ 3:
        - Generate k-bit neighbours of current_solution.
        - Shuffle the neighbours randomly.
        - For each neighbour:
            - Evaluate its fitness.
            - If its fitness is better than best_fitness:
                - Set best_solution to this neighbour.
                - Set best_fitness to its fitness.
                - Set current_solution to this neighbour.
                - Set k = 1 (reset neighbourhood size).
                - Break.
        - If no improvement found, increase k by 1.

    Return best_solution and its fitness.

Repeat the following for num_runs:
    - Generate a random initial solution.
    - Record start_time.
    - Run the variable_neighbourhood_ascent() to find the best solution.
    - Record end_time and calculate time_taken.
    - Evaluate the best solution's fitness.
    - Store the fitness, solution, and time_taken.

After all runs, report:
    - Average time over all runs.
    - Maximum clauses satisfied.
    - Average clauses satisfied.

```

Per□ormance Analysis and Results:

The algorithm was tested on the following MAXSAT instances:

- **uf20-01.cnf**: Contains 91 clauses and 20 variables.
- **uf100-01.cnf**: Contains 430 clauses and 100 variables.
- **uf250-01.cnf**: Contains 1065 clauses and 250 variables.

Results for each instance were recorded over 30 independent runs to assess the algorithm's performance in terms of solution quality, objective function evaluations, and CPU time.

For each of the three problem instances, the algorithm was executed 30 times. The following metrics were reported:

- For uf20-01.cnf (91 clauses, 20 variables), the algorithm performed well, consistently satisfying an average of 90/91 clauses across 30 runs. The maximum number of clauses satisfied in any run was 91. The algorithm was fast, with an average run time of 0.1240 seconds.
- For uf100-01.cnf (430 clauses, 100 variables), the algorithm took significantly more time, with an average of over 100 seconds per run. The maximum number of clauses satisfied

was 429/430. The increased problem size and clause complexity explain the longer execution time.

- For uf250-01.cnf (1065 clauses, 250 variables), the algorithm is taking much longer. This is due to the exponential growth in the number of neighbours generated as the problem size increases. The algorithm must evaluate a large number of neighbours, which increases the computational burden, leading to longer runtimes.

Challenges and reasoning:

The increase in runtime is primarily due to the nature of the neighbourhood generation and fitness evaluations. For each solution, the algorithm generates all possible k-bit neighbours. As the number of variables increases, the number of possible neighbours increases exponentially. For example:

- For k = 1, the algorithm generates num_vars neighbours.
- For k = 2, it generates approximately num_vars choose 2 neighbours.
- For k = 3, it generates approximately num_vars choose 3 neighbours.

Thus, as the number of variables increases, the number of neighbours grows rapidly, resulting in more time spent evaluating them.

Mathematically, if n is the number of variables, the number of neighbours generated for each value of k is:

- k = 1: O(n)
- k = 2: O(n^2)
- k = 3: O(n^3)

For larger instances like uf250-01.cnf, where n = 250, the number of neighbours becomes overwhelming, especially for higher values of k. The fitness evaluation for each neighbour also adds to the time complexity.

Time Complexity (Big O Notation)

The time complexity of the Variable Neighbourhood Ascent algorithm is determined by:

- Neighbour generation: O(n^k) for k-bit neighbours.
- Fitness evaluation: O(m) per solution, where m is the number of clauses.

For each iteration, the worst-case time complexity is:

- O(n^3 * m), where n is the number of variables, and m is the number of clauses.

Thus, the total time complexity for the entire run depends on the number of iterations and the number of neighbourhoods explored.

Multi start Variable Neighbourhood Ascent (MSVNA)

Tasks:

- Each execution of MSVNA should stop if either a global optimum is found or if by the time a given restart begins a maximum of 10 million objective function evaluations have already elapsed, whichever occurs first.
- For each instance report the best solution quality obtained, the number of objective function evaluations needed to reach that solution, and the CPU time taken to do so.

Algorithm and Pseudocode:

Neighbourhood Order: In this implementation of the Multi-Start Variable Neighbourhood Ascent (MSVNA) algorithm, we aim to find the global optimum solution for various MAXSAT problem instances.

DIMACS instances: It evaluates the algorithm with three DIMACS instances (uf20-01.cnf, uf100-01.cnf, uf250-01.cnf).

Multiple Runs Execution: The algorithm terminates early if a global optimum (i.e., all clauses satisfied) is found or if a maximum of 10 million objective function evaluations is reached.

```
Define function `evaluate_fitness(solution, clauses)`:
    - Input: solution (binary array), clauses (list of clauses)
    - Output: satisfied_clauses (integer)

Define function `generate_k_bit_neighbours(solution, k)`:
    - Input: solution (binary array), k (neighbourhood size)
    - Output: neighbours (list of neighbour solutions)

Define function `multi_start_vna(max_iterations, max_evaluations)`:
    - Initialize `total_evaluations` to 0
    - Initialize `best_global_solution` to None and `best_global_fitness` to 0

    - While `total_evaluations < max_evaluations`:
        - Generate random initial solution
        - Set `current_solution` to initial solution
        - Set `best_solution` to `current_solution`
        - Set `best_fitness` by evaluating `current_solution`

        - Set `evaluations` to 0 and `k` to 1
        - While `k <= 3` and `evaluations < max_iterations`:
            - Generate `k-bit neighbours` of `current_solution`
            - Shuffle the neighbours randomly

            - For each `neighbour` in `neighbours`:
                - Evaluate the fitness of `neighbour`
                - Increment `evaluations` and `total_evaluations`

                - If `neighbour` has better fitness than `best_fitness`:
                    - Update `best_solution` and `best_fitness`
                    - Set `improvement_found` to True
                    - Break inner loop

                - If `total_evaluations >= max_evaluations`, break outer loop

             - If `improvement_found`, set `current_solution = best_solution` and reset `k =
1`
            - Else, increment `k`

```

```
        - Update `best_global_solution` and `best_global_fitness` if `best_fitness` improves
        - If `best_global_fitness == num_clauses`, break the loop

    - Return `best_global_solution` and `total_evaluations`

Define main procedure:
    - Set parameters (`num_runs`, `max_iterations`, `max_evaluations`)
        - Initialize results lists (`times`, `best_solutions`, `best_fitness_values`, `evaluations_list`)

    - For `run` in range `num_runs`:
        - Start timer
        - Call `multi_start_vna` and record best solution, total evaluations, and time taken
        - Record fitness and update lists

    - Print results summary (average time, max satisfied clauses, etc.)

```

Performance Analysis and Results:

The algorithm was tested on the following MAXSAT instances:

- **uf20-01.cnf**: Contains 91 clauses and 20 variables.
- **uf100-01.cnf**: Contains 430 clauses and 100 variables.
- **uf250-01.cnf**: Contains 1065 clauses and 250 variables.

For smaller instances like uf20-01.cnf, the MSVNA algorithm consistently finds the global optimum (91/91 satisfied clauses) in a relatively short amount of time (0.3244 seconds on average). For larger instances like uf100-01.cnf and uf250-01.cnf, the algorithm takes significantly longer and struggles to find the optimum as efficiently.

Reasons for Performance Differences: The observed increase in time and number of evaluations for larger instances is expected due to the following factors:

Search Space Size:

- The number of possible solutions (search space) grows exponentially with the number of variables. For example:
  - uf20-01.cnf has 20 variables → 2202^{20}220 possible solutions.
  - uf100-01.cnf has 100 variables → 21002^{100}2100 possible solutions.
  - uf250-01.cnf has 250 variables → 22502^{250}2250 possible solutions.
- Larger instances like uf100 and uf250 have exponentially larger search spaces, making it more difficult to find optimal solutions quickly.

Neighbourhood Size and Exploration:

- The algorithm explores neighbours by flipping up to 3 bits at a time. For larger instances, generating and evaluating these neighbourhoods takes longer as the number of neighbours increases with the number of variables.

- The k-bit neighbourhood size affects how thoroughly the local area around a solution is explored. Larger neighbourhoods are required to escape local optima, which is more challenging in larger instances.

Randomness in Metaheuristics:

- Metaheuristic algorithms like MSVNA depend on randomness, both in initial solution generation and in the order of neighbourhood exploration. For larger instances, this randomness introduces variability in the time taken to find a good solution.

Big O Complexity:

The time complexity of this MSVNA implementation can be broken down as follows:

- **Initial Solution Generation:** O(n), where n is the number of variables.
- **Neighbourhood Generation:** For each solution, the number of k-bit neighbours generated is $\binom{n}{k}$, which is O(n^k). Since k ranges from 1 to 3, the neighbourhood generation for each iteration is O(n^3).
- **Fitness Evaluation:** Evaluating each neighbour requires O(m) time, where m is the number of clauses. Therefore, each fitness evaluation for a neighbour takes O(m) time.

Given that the total number of objective function evaluations is capped at 10 million, the overall time complexity per run can be approximated as:

$$Time\ Complexity \approx \mathrm{O}(\mathrm{T} \times n^3 \times \mathrm{m})$$

where T is the total number of evaluations.

For larger instances (like uf250-01.cnf), n and m are much larger, leading to a significant increase in the time complexity, which explains the longer CPU times.

The MSVNA algorithm performs well on small to moderate-sized MAXSAT instances, quickly finding optimal solutions. However, for larger instances, the computational effort increases substantially due to the exponential growth of the search space and the time required to explore larger neighbourhoods. This highlights the trade-off between solution quality and computational efficiency in metaheuristic approaches like MSVNA.

References:

1) 1.5.2 Time Complexity Example #2 by Abdul Bari. https://www.youtube.com/watch?v=9SgLBjXqwd4
2) A comprehensive cheat sheet that explains Big O notation and provides examples of common complexities. https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/
3) A guide to calculating Big O time and space complexity, including examples and explanations. https://www.inoutcode.com/concepts/big-o/
4) A step-by-step guide to calculating Big O notation, including identifying input size and counting operations. https://blog.algorithmexamples.com/big-o-notation/step-by-step-guide-calculating-big-o-notation/