Algorithms Programming Project I
Greedy Algorithms

Team Members

Gurazeez Singh Sachdeva - 48747105
Algorithm Analysis (time complexity and optimal solutions), Greedy strategies, Task implementation for different strategies, Experiment analysis.

Justin Strikowski - 52181920
Experimental analysis, Task implementation, and testing different input cases, debugging, makefile, technical writing

Problem Definition

You are a house painter that is available from day 1 . . . n (inclusive). You can only paint one house in a day. It also only takes one day to paint a house. You are given m houses. For each house i, you are also given startDayi and endDayi for i = 1, . . . , m. The house i can only be painted on a day between startDayi and endDayi (inclusive). The given houses are already sorted primarily on startDay and secondarily on endDay (in case of equality of the startDay). You are tasked to find the maximum number of houses that you can paint.

Strategy 1 - Earliest Available First:

Iterate over each day starting from day 1 . . . n. For each day, among the unpainted houses that are available to be painted on that day, paint the house that started being available the earliest.

Algorithm:

```
1. Initialize variables "painted" and “currentDay" to 0 and 1, respectively.
2. For each house:

    a. If startDay > currentDay, increment currentDay to startDay.

    b. If currentDay <= endDay and currentDay >= startDay, paint the house i,
    increment painted by 1, and increment currentDay by 1.

3. Return the list of painted houses.
```

Time Complexity: O(n)

Explanation:

The algorithm simply iterates over each day and paints the house that started being available the earliest among the unpainted houses available to be painted that day. Since the houses are sorted primarily on startDay, the earliest available house will be the first one encountered. The maximum number of times that the loop can iterate is the number of days, since the code skips all days where there is not a house. Instead, it will skip to when the next available house starts. So, there cannot be an instance of the loop run where there is not a house either removed or inputted. The time complexity of this algorithm is O(n), where n is the number of days.

An instance where the strategy yields an optimal solution:

Suppose we have 5 houses with the following start and end days:
House 1: startDay=1, endDay=5
House 2: startDay=1, endDay=3
House 3: startDay=3, endDay=6
House 4: startDay=4, endDay=7
House 5: startDay=6, endDay=8

Using Strategy 1, the optimal solution is to paint each house on its respective day, which strategy 1 yields.

An instance where the strategy does **not** yield an optimal solution:

Suppose we have 3 houses with the following start and end days:
House 1: startDay=1, endDay=3
House 2: startDay=1, endDay=3
House 3: startDay=2, endDay=2

Using Strategy 1, the algorithm will paint houses 1 on day 1 and house 2 on day 2. It will not paint house 3, because it is only available on day 2. However, the optimal solution is to paint only houses 1 on day 1, house 3 on day 2, and house 2 on day 3. Note that houses 1 and 2 can switch days. Because the strategy does not yield the optimal solution of painting all three houses in three days, it is proved to be not optimal by counterexample.

Strategy 2 - Latest Start First:

Iterate over each day starting from day 1 . . . n. For each day, among the unpainted houses that are available that day, paint the house that started being available the latest.

Algorithm:

1. Sort the houses list by start day (in ascending order) and reverse end day so that the earliest end day is first in the stack.
2. Initialize a stack variable to an empty list and currentDay to 0.
3. Loop through each day from 1 to n.
    a. While there are unpainted houses that become available on this day, pop them off the houses list and append them to the stack as tuples of the form (start, end, index).
    b. While the stack is not empty:
        i. Pop off the last added house from the stack as (start, end, index).
        ii. If the end day is less than the current day, skip to the next iteration of the current, small loop. This essentially removed that house from the stack as it is now invalid and will never be reached
        iii. If the house is still available, print the index of the house and break out of the loop.

Time Complexity: O(n + m log(m)) where n is days and m is houses.

Explanation:

The algorithm first sorts the houses in descending order of start day, which takes m*log(m) time using quicksort. Stack section of the for loop cannot run more than 2m times because once a house is added to the stack (which can happen only once), it is removed only when painted, so the number of stack insertions/deletions is at most 2m, where the stack insertion is ran at MOST two times for each house. Each insertion/deletion is O(1), so that total is 2m, which leads to O(n + m + m*log(m)), but the m is canceled out by the m*log(m) of the initial sort. Adding those together, the total complexity of the algorithm that I wrote is O(n + mlog(m)).

An instance where the strategy yields an optimal solution:

Suppose we have 4 houses with the following start and end days:
House 1: startDay=1, endDay=3
House 2: startDay=2, endDay=4
House 3: startDay=3, endDay=5
House 4: startDay=4, endDay=6

Using Strategy 2, the algorithmic solution is to paint houses 1, 2, 3, and 4 on their respective days, for a total of 4 houses. This is the optimal solution.

An instance where the strategy does **not** yield an optimal solution:

Suppose we have 3 houses with the following start and end days:
House 1: startDay=1, endDay=2
House 2: startDay=1, endDay=2
House 3: startDay=2, endDay=3
House 4: startDay=2, endDay=4

Using Strategy 2, the algorithm will paint house 1 on day 1, house 3 on day 2, house 4 on day 3, and no houses on day 4. This only paints 3 houses, whereas the optimal solution would paint each house on its respective day. Therefore, we proved that the latest start time algorithm is not optimal by counterexample.

Strategy 3 - Shortest Duration First:

Iterate over each day starting from day 1 . . . n. For each day, among the unpainted houses that are available that day, paint the house that is available for the shortest duration.

1. Initialize a list of tuples called houses with elements (start, end, i+1) where i is the index of the house and start and end are the start and end day of the house.
2. Sort the houses list by start day and reverse end day.
3. Iterate over days from 1 to n.
    a. While there are unpainted houses and the first house in houses list can be painted on the current day, add the house to the available heap.
    b. While there are houses available, pop the house with the shortest duration and paint it if it can be painted on the current day. Add the index of the painted house to the painted list and break out of the loop.
4. Print the indices of the painted houses in painted.

Time Complexity: O(n + m log(m))

Explanation:
Sorting the list of houses by start day and reverse end day takes m*log(m) time. The algorithm has a time complexity of O(n + m*log(m)) because the loop runs n times and the insertion into the binary heap takes log(m) time. However, the binary heap section of the code cannot run more than 2m times because once a house is added to the heap (which can happen only once), it is removed only when painted, so the number of heap insertions/deletions is at most 2m, where the heap insertion is ran at MOST two times for each house. Each insertion/deletion is log(m), so that total is 2m*log(m), which is canceled out by the m*log(m) of the initial sort. Thus, the overall time complexity of the algorithm can be simplified to O(n + m*log(m)) where n is the number of days and m is the number of houses.

An instance where the strategy yields an optimal solution:

Suppose we have 4 houses with the following start and end days:
House 1: startDay=1, endDay=1
House 2: startDay=1, endDay=2
House 3: startDay=3, endDay=4
House 4: startDay=3, endDay=3

Using Strategy 3, the optimal solution is to paint houses 1, 2, 3, and 4, for a total of 4 houses. The algorithm will paint house 1 on day 1, house 2 on day 2, house 4 on day 3, and house 3 on day 4. The strategy prioritizes the house that is available and has the shortest total availability, which leads to house 4 being painted before 3. House 2 is not painted before 1, though, since 2 has a larger window.

An instance where the strategy does **not** yield an optimal solution:

Suppose we have 4 houses with the following start and end days:
House 1: startDay=1, endDay=3
House 2: startDay=1, endDay=2
House 3: startDay=2, endDay=3
House 4: startDay=3, endDay=4

Using Strategy 3, the algorithm will paint houses 2, 3, and 4, for a total of 3 houses. However, the optimal solution for this instance is to paint houses 2 and 3 on days 1 and 2, house 1 on day 3, and house 4 on day 4 for a total of 4 houses. The algorithm fails to paint house 1 since the other houses all have higher precedents since their availability is shorter. The kicker is that 1 could have been painted on day 3, but since 4 only has a 1 day duration, it got priority for that day, leaving house 1 to remain unpainted on day 4. Therefore, because this algorithm does not yield the optimal solution, it is proved to be false by counterexample.

Strategy 4 - Earliest Finish Time First:

Iterate over each day starting from day 1 . . . n. For each day, among the unpainted houses that are available that day, paint the house that will stop being available the earliest.

Algorithm:

1. Sort the houses based on start day and end day (earliest start day first, then earliest end day first)
2. Initialize an empty priority queue, a list to store whether a house is painted or not, and an output list
3. Iterate over the days from 1 to n
    a. Add all the unpainted houses that are available on that day to the priority queue
    b. Paint the house that will stop being available the earliest among the unpainted houses available on the current day. If a house is found to be no longer available, remove it from the list.
    c. If a house is painted, mark it as painted in the list and remove it from the list of houses to paint
    d. Print the index of the painted house

Time Complexity: O(n + m log(m))

Explanation:

The algorithm iterates over each day and paints the house that will stop being available the earliest among the unpainted houses available to be painted that day. The houses are sorted at the beginning, which takes mlog(m) time. The binary heap insertion/deletion cannot run more than 2m times because once a house is added to the heap (which can happen only once), it is removed only when painted, so the maximum number of heap operations 2m operations of log(m) (m*log(m), in total), where the heap insertion is ran at MOST two times for each house. Thus, the overall time complexity of the algorithm can be simplified to O(n + m*log(m)) where n is the number of days and m is the number of houses.

An instance where the strategy yields an optimal solution:

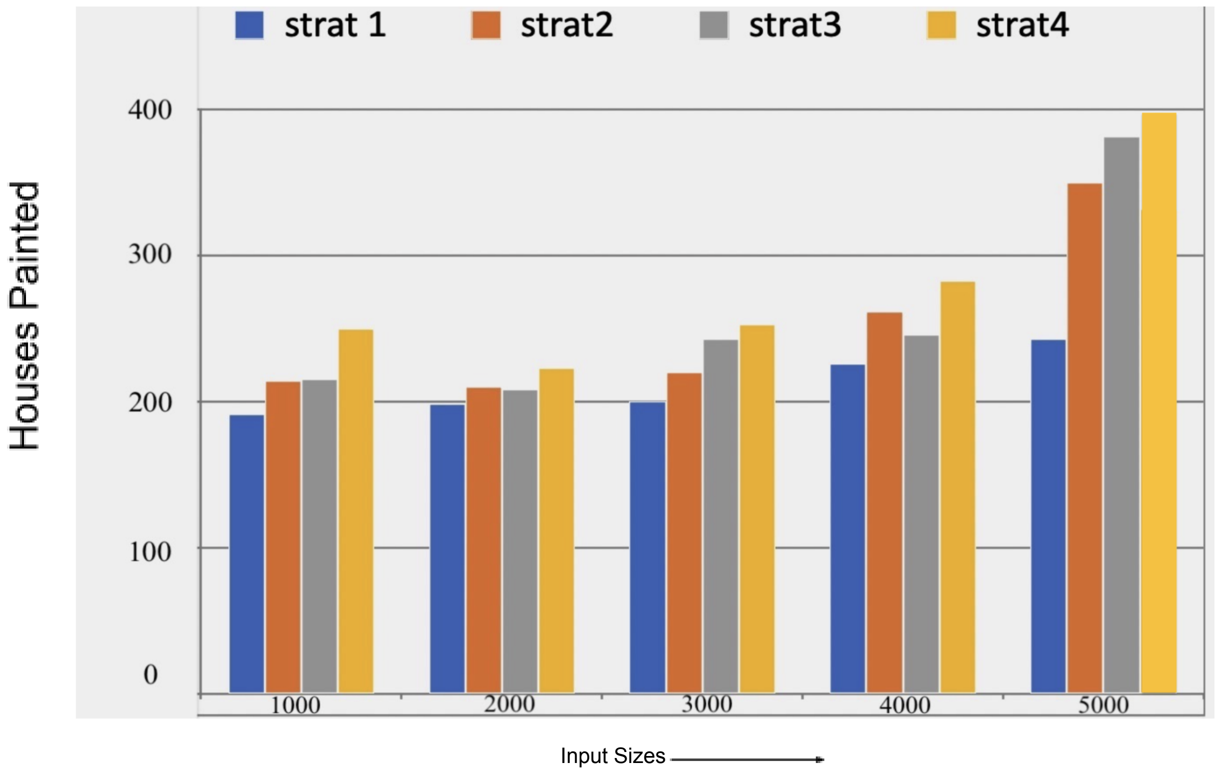
Suppose we have 4 houses with the following start and end days:
House 1: startDay=1, endDay=4
House 2: startDay=2, endDay=5
House 3: startDay=3, endDay=6
House 4: startDay=3, endDay=3

The optimal solution can paint all of the houses, house 1 on day 1, house 2 on day 2, house 4 on day 3, house 3 on day 4. Strategy 4 achieves this.

An instance where the strategy does **not** yield an optimal solution:

Since 4 is the optimal algorithm, it always yields an optimal solution. I will prove that it will always yield the optimal solution by contradiction.

- Assume that the earliest-finish-time-first is not optimal.
- Let i1….ik be the greedy selection of houses
- Let j1….jm be the optimal selection of houses with i1=j1…ir=jr for the largest r.
- If j+1 does not interfere with jr+2 nor ir+1 or 2, that means that there exists another solution, but that solution contradicts the maximality of the value r
- Therefore, by contradiction, the greedy earliest-finish-time-first algorithm is optimal!



Above is a graph of the time taken for each of the implementations for 1k-5k inputs. As you can see, the strategy 4, the earliest-finish-time-first-algorithm, is consistently the highest performing algorithm. This is because it is the optimal solution to paint the most houses in a given time, as proved prior. This greedy algorithm produced the optimal solution for interval scheduling problems. The first algorithm, on the other hand, consistently performed poorly, especially as the sample size increased. There were too many times in that algorithm where houses that were going to soon become unavailable were not prioritized because the algorithm sorted by start time. The simplicity of this algorithm, while it is beneficial for the time complexity, takes its toll in the efficacy category. The second and third algorithms, the latest start time algorithm and the shortest availability algorithm respectively, both perform similarly in terms of efficiency. They may miss out on certain houses that could be painted, resulting in a slightly suboptimal solution. While they are not as optimal as the earliest-finish-time-first algorithm, they are still much more effective than the first algorithm.

Conclusion:

Creating the solutions was easy. What was difficult was creating solutions that were *optimal*. That is, implementations that have the time complexity that is required in the description. Additionally, we discovered hiccups late on that required me to redo three of the functions. The problem was that the test cases that were posted on the discussion were not comprehensive. So, as soon as we passed the test cases, we moved onto the analysis. It was a mistake for assuming that they were comprehensive, but having to redo most of the analysis was quite the hassle.

The first task was the easiest to implement. Since it functions in O(n) time, a simple loop that looped over the days was sufficient for implementing the algorithm. The implementation of this took less than 45 minutes including debugging, which is a small number compared to the rest of the algorithms.

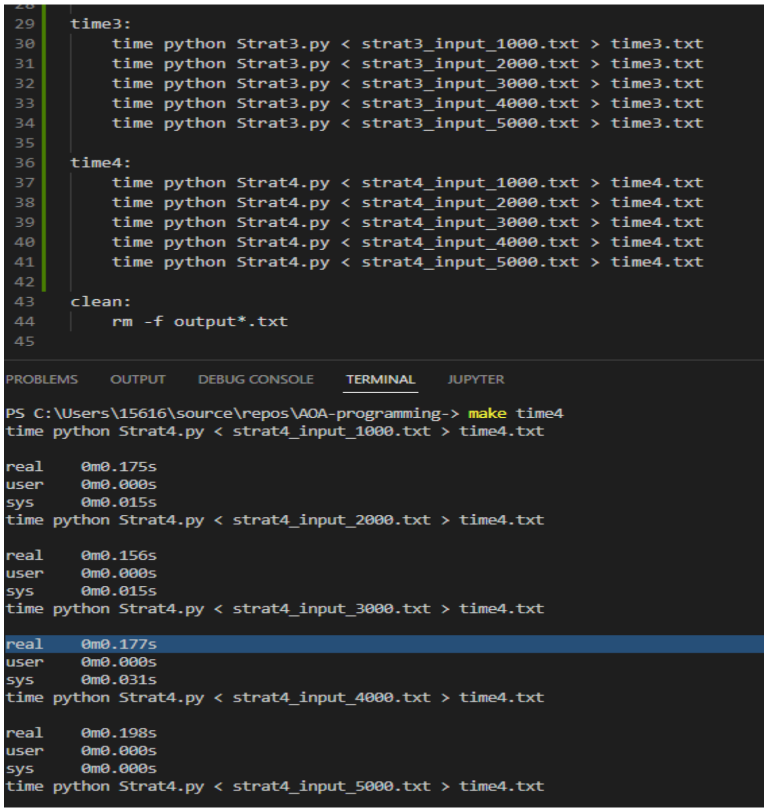
The second task was the toughest. The first implementation took an hour and passed all of the test cases. A while later, We decided to make more test cases that disproved the correctness of my first implementation. It was upsetting that we had to go back to the drawing board. A quick look at our github repository will show that the second task took a lot more effort, commits, and time to implement than all of the others. However, once we created my own test cases, those greatly helped with debugging and made the code easier to fix.

The third task was only slightly easier than the second. The main problem that we had was that my initial implementation did not use a queue, but instead it used a simple array, which did not allow for a priority for shortest duration first. We also had to alter the parameters to subtract the start from the end, so that was my vital first step in fixing my rough draft implementation of my pseudocode. Additionally, we found a way to doubly sort the houses array, which allowed us to implement an optimal solutions instead of just one that satisfies the requirements. Making it more optimal made us feel better about it, as it outperformed some of the test cases!

This one fell in line after creating the third case. The line that doubly sorts the array that I used for the third algorithm immediately made my implementation easier. The pseudocode was not working, so we went back to the third case to work on that. Once we found the sorting line, we applied it to strategy 4 and it nearly fixed the entire algorithm. The implementation of the heap queue was added later on to reduce the time complexity. That was the main challenge with the fourth case: getting the time complexity down to what it required in the instructions. Once that was added, implementing my own test cases proved my initial implementation wrong *again*. Luckily, we just had to put the end at the beginning of the heap push instead of in the middle. That way, it sorts by the finish time. we would not have been able to find that without my own test cases.

The moral of the story is: code with test-driven development. Incorrect code is no good. Thorough test cases are the only way to ensure that a program is correct. Additionally, the test cases (specifically edge cases) were the primary thing that led to debugging the dysfunctional algorithms. This assignment, more than any other, taught me the lesson: make thorough test cases!

Also, the graphing posed a challenge initially. I tried to do an automatic compilation, execution, and graph in a Jupyter notebook all at the same time. However, this ended up having too many moving parts that did not work together. Instead, we took the data that was generated initially by the Jupyter file and added it to the repository directory. Then, I used the time function in the command line to time the function (I did not know that that could be done! I used the makefile to compile multiple at the same time. The results are shown below. There is another lesson learned, sometimes the simpler method is quicker, since it will take less time to figure out!



```
time3:
    time python Strat3.py < strat3_input_1000.txt > time3.txt
    time python Strat3.py < strat3_input_2000.txt > time3.txt
    time python Strat3.py < strat3_input_3000.txt > time3.txt
    time python Strat3.py < strat3_input_4000.txt > time3.txt
    time python Strat3.py < strat3_input_5000.txt > time3.txt

time4:
    time python Strat4.py < strat4_input_1000.txt > time4.txt
    time python Strat4.py < strat4_input_2000.txt > time4.txt
    time python Strat4.py < strat4_input_3000.txt > time4.txt
    time python Strat4.py < strat4_input_4000.txt > time4.txt
    time python Strat4.py < strat4_input_5000.txt > time4.txt

clean:
    rm -f output*.txt
```

```
PS C:\Users\15616\source\repos\AOA-programming-> make time4
time python Strat4.py < strat4_input_1000.txt > time4.txt

real    0m0.175s
user    0m0.000s
sys     0m0.015s
time python Strat4.py < strat4_input_2000.txt > time4.txt

real    0m0.156s
user    0m0.000s
sys     0m0.015s
time python Strat4.py < strat4_input_3000.txt > time4.txt

real    0m0.177s
user    0m0.000s
sys     0m0.031s
time python Strat4.py < strat4_input_4000.txt > time4.txt

real    0m0.198s
user    0m0.000s
sys     0m0.000s
time python Strat4.py < strat4_input_5000.txt > time4.txt
```

BONUS:

The greedy algorithm that prioritizes the earliest finish time is optimal.
An m*log(m) algorithm that would implement this would look like:

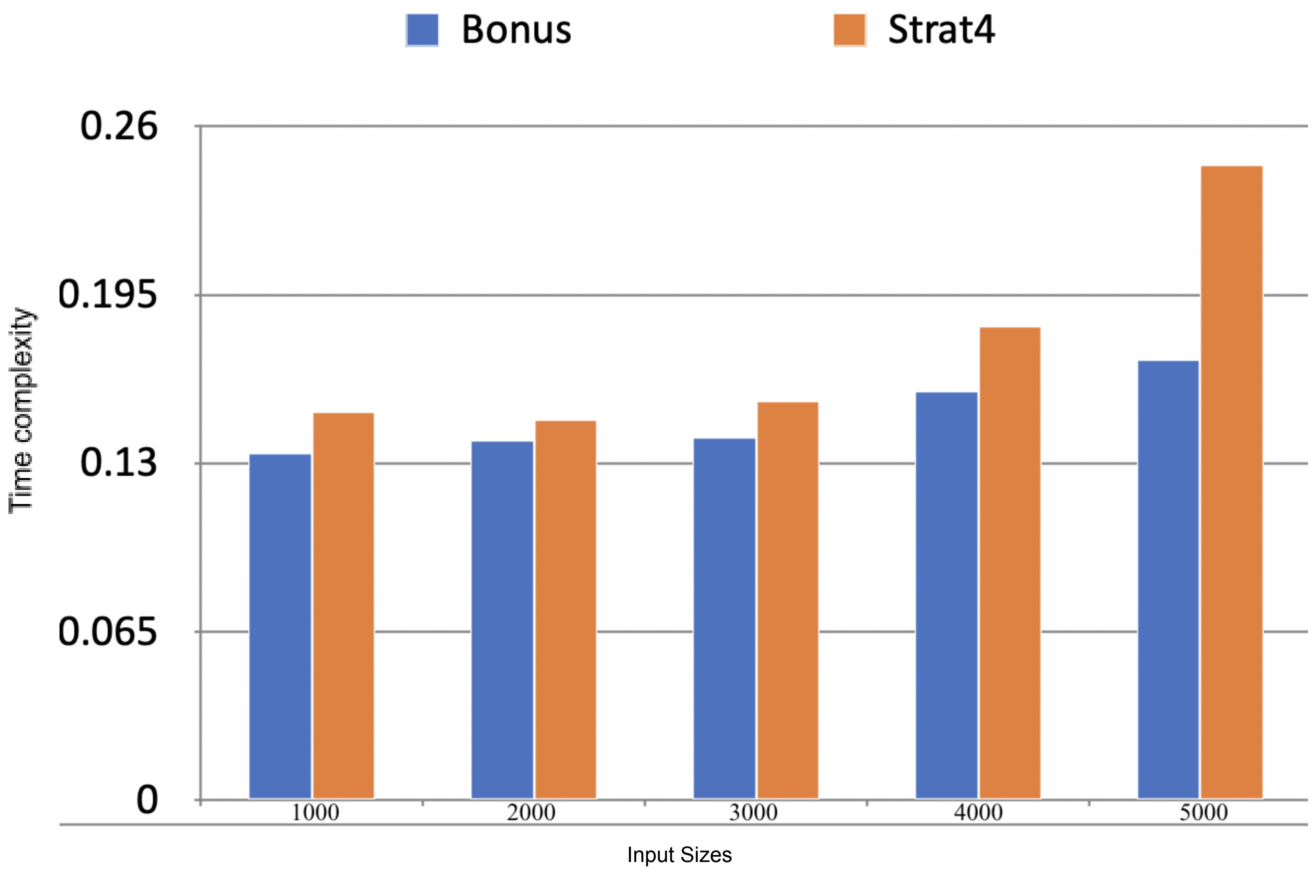
- Sort the houses by earliest start time (and if there is a tie, it is broken by the earliest finish time. (This takes 2m*log(m, which, for big theta, reduces to theta(m*log(m))
- Set day to 1
- Add houses to a priority queue that prioritizes finish time if the day is 0 (will sum to theta(mlogm) because insertion for each element takes logm, so mlogm

- Loop over the queue of houses and print ones that are available and have the earliest finish time of the available houses. If the house availability has already ended, remove it from the queue

The initial sorting takes mlog(m) time. The prioritization of each house also takes log(m) for each house that is added to the available list. Because each operation in this algorithm is logm (or logm run m times, so mlog(m)), the total algorithmic complexity of the algorithm is O(mlog(m)). When run in parallel with the original implementation of part 4, the bonus mlog(m) implementation took, consistently, less than 75% of the execution time!

I must say it after the conclusion, because this does not make sense to include in the conclusion, but the bonus was one of the hardest to get right.

Time complexity bonus:



As shown in the graph, the O(mlog(m)) graph consistently outperforms the O(n + mlog(m)) from strategy 4. It should be noted that this difference becomes more exaggerated with larger inputs, because the n is tacked on to the end, which results in the staggering difference that you see on the right, with the 5000 house input.