Josh Dow

April 21, 2019

1 Showing a Solution Exists

Claim: For any given instance $h_1, h_2, ...h_n$ of this problem satisfying the above input conditions, there is a sequence of n real numbers $b_1, b_2, ...b_n$ such that $0 \leq b_1 < b_2 < ... < b_n$ and for every integer $1 \leq i \leq n$ there is an integer $1 \leq j \leq n$ such that $|h_i - b_j| \leq 4$.

Proof. Since there are both n houses, and n towers, a simple solution that will always exist is to set b_j to h_i for each h_i in $0 \leq i \leq n$. This means that $|h_i - b_j| = |h_i - h_i| = 0 \leq 4$. □

2 Showing that every instance has a Correct Solution

2.1 A Measure Function for Solutions

Claim: Given a list of b towers, $b_1, b_2, ...b_n$ as a solution to the tower placement problem. The function $f(b) = |b| = n \in \mathbb{N}$ is a measure function for the solution.

Proof. Proof by Construction. The above function is total for all possible outcomes. Since there are will always be at least one solution as shown in section 1, this measure function may be used, however there may be infinite solutions for any given instance. Since every outcome is a set of towers to be built $b_1, b_2, ...b_n$, taking the length of the outcome will map every possible one to a natural number. Since the natural numbers can be ordered, the outcomes may be ordered as well. This means that one of the outcomes can be decided to be minimal through the use of this measure function. Since $f([b_1, b_2, ...b_n]) = n$ is total and allows the ordering of outcomes in terms of real numbers (in this case natural), it can be said to be a measure function. □

2.2 Solutions for Trivial Instances

Claim: For a trivial instance $n = 0$, a correct solution exists.

Proof. Given an empty set of houses ($\emptyset$), no towers are required to serve the houses. This means that the solution with the least number of towers is the one with no towers, returning $\emptyset$ meaning the outcome with the shortest length has length $|\emptyset| = 0$. □

Claim: For a trivial instance $n = 1$, a correct solution exists.

Proof. For a set of houses $[h_1]$, a single tower is required to serve the full set. This means that simply returning the original houses location as a set will complete the algorithm. Therefore a solution will simply be $[b_1]$ with $b_1 = h_1$. □

2.3 A Solution for Every Instance

Claim: A solution exists for every instance of the probem.

Proof. It was previously proved that every instance of the problem has at least one potential solution. This, combined with the measure function described above, means that the set of potential solutions can be mapped to a nonempty set of Natural numbers. This can be used with the Least Number Principle, which states that evey nonempty subset of the set $\mathbb{N}$ has a smallest element, to show that the set of possible solutions will have a smallest element. This smallest element is the result of the mapping of a correct solution through the measure function. Every one of these minimum potential solutions is a correct solution to the to the given instance. □

3 Designing and Proving Correctness of a Greedy Algorithm

3.1 Designing the Algorithm

Claim: By checking if each home is already covered, and if not, placing the tower at maximum range from the home along the road, a greedy algorithm can be created.

Proof. This will be done through a exchange argument. Given the input $[h_1, h_2...h_n]$, and that a solution always exists, suppose that a solution of the form $[b_1, b_2, ...b_k]$ exists. Then there exists a equivalent solution such that b_1 is replaced with: $h_1 + 4 = b'_1$. This solution is equivalent in length, as the previous one, with coverage over the first house h_1. There are no houses that may be covered by b'_1 by moving it any closer to the origin/ h_1, and placing it any further would leave h_1 without coverage. Therefore this would be the correct placement.

The next step in the proof is to find all of the remaining houses that lack coverage. This is essentially a smaller instance of the same problem. Which means that it can be solved in a similar fashion. Given the set $[h_i, h_{i+1}, h_n]$ of uncovered houses with the solution $[b_j, b_{j+1}, ...b_k]$ the same process as the first

part can be repeated, replacing b_j with $b'_j = h_i + 4$. Which as above, can be shown to be the correct placement. This subproblem solving is repeated until there are no uncovered homes.

Once no more towers are needed ($\emptyset$ is given as the input), simply take all of the tower $[b'_1, ...b'_j, ...b'_k]$ which will be the solution for the problem $[h_1, h_2...h_n]$. □

3.2 The Algorithm

```
float[] plantTowers(float[] houses, float radius) {
        if(houses.length == 0){
            return new float[0];        //Empty array
        }
        else if (houses.length == 1){
            return houses;
        }
        List towers = new List<float>();
        float furthestReach = -1;
        for(int i = 0; i < houses.length; i++){
            if(houses[i] > furthestReach){
                towers.append(houses[i]+radius);
                furthestReach = houses[i] + 2*radius;
            }
         }
        return towers.toArray();
}
```

3.3 Proving Correctness

3.3.1 Loop Invariant

Claim: The following are all loop invariants for the above algorithm:

1. The precondition for the "Base Station Placement" problem is satisfied.
2. $houses.length \geq 2$
3. $towers$ is an list of floats with length $\leq i$
4. $0 \leq i < houses.length$
5. $houses[j]$ is has coverage for all $j < i$
6. The least number of towers is used to generate coverage up to $houses[i-1]$

Proof. A small (one sentence) proof for each of the above is matched by number below:

1. Since the components of the "Base Station Placement" problem (houses array, radius) are not modified at any time of the loop execution, this is true given the preconditions were correct to begin with.

2. Since the test on either line 2 or 5 would pass given an array with length less than 2, the function would return on line 3 or 6 respectively, meaning the loop would not be executed. Therefore to reach the initial loop the length would have to be at least 2. This holds for all required portions of the loop invariant as the array is never modified.

3. Since the type of *towers* is never changed, it will always be an *List*, no matter the location in the loop. As for the length, for the worst case scenario, the loop can only add 1 item per iteration, this limits the length of the list to be less than or equal to i for all locations.

4. The iterator i, which is initialized to 0, will always be between that, and the length of the array (minus one due to indexing).

5. If a house $houses[j]$ with $j = i$ lacks coverage, then it will pass the test on line 11, which appends a new tower to the list *towers* granting the tower coverage. Therefore it is not possible to increment i so $j < i$ while maintaining no coverage on $houses[j]$.

6. As previously seen in the construction of the greedy method, the minimum number of towers are used to cover up to $houses[i-1]$.

□

3.3.2 Partial Correctness

Claim: The *plantTowers* algorithm is partially correct, when considered as an algorithm for the "Base Station Placement" problem.

Proof. Consider an execution of this algorithm with the precondition for the "Base Station Placement" problem satisfied.

Where $n = houses.length$.

If $n = 0$, steps 2 and 3 are executed and it certainly returns the required output after execution.

If $n = 1$, step 2 is executed and fails and then steps 5 and 6 are executed. It also certainly returns the required output after execution.

On the other hand, if $n \geq 2$, the test at lines 2 and 5 fail, then execution begins with the step at line 8. Lines 8 and 9 are executed as expected and the loop is reached. This will then be iterated n times. It can be seen by inspection of the coed that the variables *houses* and i are not modified, so the amount of times the loop will be executed cannot be changed from within the loop.

This means it is finite and the execution of the loop will eventually end. This satisfies the loop invariant $0 \leq i < houses.length$. Then once line 16 is reached and executed the correct output is then returned.
It follows that this algorithm is partially correct as claimed. □

3.3.3 A Bound Function

Claim: $f(houses.length, i) = houses.length - i$ is a bound function for the above pseudo-code.

Proof. This will be shown through a proof by construction. The first requirement of a bound function is that it is integer valued. Here $houses.length$ is an integer derived from the input length and i is an integer variable that is created at the start of the for loop. Since i is increased by one and $houses.length$ is unchanged when the loop body is executed, the function's value decreases by one during every execution of the loop body. The second requirement is that the loop halts if the value is of the function is less than or equal to 0. If this function's value is less than or equal to 0 then $i \geq houses.length$ and the test at line 10 fails. This forces the termination of the loop. □

3.3.4 Termination

To prove that the termination one must prove two things:

1. If the problem's precondition is satisfied when an execution of the algorithm including this loop begins, then every execution of the loop body ends.

2. A bound function for the loop exists.

Claim: When the *plantTowers* algorithm is executed with the precondition for the "Base Station Placement" problem satisfied, then this execution terminates.

Proof. Consider an execution of this algorithm with the precondition for the "Base Station Placement" problem satisfied. An array of floats *houses* and a float *radius* are given as input.

Where $n = houses.length$.

If $n = 0$, steps 2 and 3 are executed and it certainly terminates after execution.

If $n = 1$, step 2 is executed and fails and then steps 5 and 6 are executed. It also certainly terminates after execution.

For all $n \geq 2$, steps 2, 5, 8, 9 are executed and then the first loop is reached at step 10 and executed. The body of this loop has only 3 steps and they never effect the conditions of the loop. Since the loop test is based on $i < houses.length$,

those are the values that would need to be affected so that the loop doesn't end. We can see by inspection of the code that the only variables that could be modified are $towers$ at step 10 and $furthestReach$ at step 11. So this loop certainly terminates whenever it is executed. It also follows by the second part of the definition, which is that the loop has a bound function. This bound function is $houses.length - i$ as shown in the previous question.

After this loop has terminated, the algorithm will surely terminate since the only line left to execute is the return statement at line 16.

□

3.3.5 Total Correctness

Claim: When the *plantTowers* algorithm is executed with the preconditions for the "Base Station Placement" problem satisfied, the algorithm is correct.

Proof. This will be shown through a proof by construction. An algorithm is correct if it is partially correct and terminates when the preconditions for the algorithm are satisfied.

It has been proven in previous questions above that the algorithm, *plantTowers*, is partially correct and that terminates. In our claim we execute the algorithm with the preconditions for the "Base Station Placement" problem satisfied. Due to the consequence of the definitions, this algorithm is correct. □

4 Upper Bound of Running Time

4.1 Cost Assumptions for Running Times

On top of the uniform cost criterion, several additional assumptions must be made before the running time for the algorithm can be discussed:

1. The initialization of an empty array can be done in c_1 steps
2. The initialization of an empty list can be done in c_2 steps
3. Appending a value onto the end of the array can be done in c_3 steps (assuming a doubly linked list)
4. Converting a list to an array can be done in $c_4 \times n + c_5$ steps
5. Floating point arithmetic has unit cost

With these assumptions in place it is now possible to begin to find the running time for the above algorithm.

Given an input of length 0, the above algorithm executes the test at line 2, which after passing, will generate an empty array and return it on line 3. The execution then ends. This requires 1 step for the test and c_1 to create an empty array.

The next execution is if the length of the input array is 1. Here the algorithm executes the test on line 2, which after failing, leads to the execution of the test on line 5, which after passing ends the execution by returning on line 6. A total of 3 steps are required.

Finally for any array of length greater than 1, two steps are required for the tests on lines 2 and 5. Then c_2 steps to initialize the empty list on line 8, followed by 1 to initialize the float on line 9.

In the worst case the interior of the loop executes 3 lines (11,12,13) with costs 1, c_3, 1. The loop itself is executed n times, as it checks each index in the houses array. This means the cost of the loop including guard checks is $n(c_3+3)$. After this one additional check is required to fail, along with the cost of changing the list to an array $c_4n + c_5$ in the worst case where every house requires its own tower. This brings the cost for $n > 1$ to $2 + c_2 + 1 + n(c_3 + 3) + 1 + c_4n + c_5$. This can be simplified to $(c_3+c_4)n+c_2+c_5+4$. These can be used to construct a recurrence for the running time as seen below:

4.2 A Recurrence

$$T(n) = \begin{cases} c_1 + 1 & \text{if n = 0} \\ 3 & \text{if n = 1} \\ (c_3 + c_4)n + c_2 + c_5 + 4 & \text{if n>1} \end{cases}$$

It can be seen from this that the fastest growing term for the recurrence is in the order of $O((c_3 + c_4)n)$ the next step is to prove that $T(n) \in O(n)$

4.3 The Limit Test

Claim: The recurrence $T(n) = \Theta(n)$.

Proof. This will be done by construction. A function $f(n) = \Theta(g(n))$ if the limit as n approaches infinity of $\frac{f(n)}{g(n)} = c$ for some constant $c > 0$. The given $g(n)$ for this example will be $g(n) = n$, with the function $f(n) = T(n) = (c_3 + c_4)n + c_2 + c_5 + 4$ for significantly large n.

$$\begin{aligned} \lim_{n\to\infty} &= \frac{f(n)}{g(n)} \\ \lim_{n\to\infty} &= \frac{(c_3 + c_4)n + c_2 + c_5 + 4}{n} \\ \lim_{n\to\infty} &= \frac{(c_3 + c_4)n}{n} = c_3 + c_4 \end{aligned}$$

Since the limit as $n \to \infty$ is a constant number $c_3 + c_4 > 0$, the recurrence $T(n)$ showing the run time of the algorithm is tight bounded by $\Theta(n)$. This means that the algorithm is also a member of $O(n)$ and $\Omega(n)$ □