HW #3 - Problem 3

Ryan St.Pierre (ras70)

September 27, 2017

Problem 3A

If the coordinates are sorted then the placing of towers can be done in linear time. This can be done by looping through the coordinates and placing a tower each time we reach a coordinate that is out of reach from a previous cell tower. When a coordinate/house that is not covered is reached, a cell tower can be placed 4 miles to its right since this will still cover the coordinate and give the best chance that the dropped tower covers points further to the right (assuming iteration through the coordinates is from *left* to *right*). These algorithmic steps are detailed below:

1. Sort the coordinates in non-decreasing order, that is $x_i \leq x_{i+1}$ for all i in 1...n.
2. Find the first instance of a house that is not in range of the currently placed cell towers. Here, *first* means the left-most house (smallest coordinate value). In other words, find the house that is more than 4 miles to the right from the last dropped tower. On the first iteration of these steps the first instance of a house that is not in range will correspond to the house at x_0.
3. Place a tower 4 miles to the right of the house found in Step 2. In other words, if house j is chosen in Step 2, place a tower at $x_j + 4$.
4. Repeat Step 2 and 3 until all houses have been covered.

To make the steps above more concrete pseudo-code has been included below.

```
dropTowers(m, n, x) {

    x = mergeSort(x)

    i = 0;

    while i < n {

        lastTower = x[i] + 4
        drop tower at lastTower

        i++

        // find the set of coordinates that the last tower covers
        while (i<n && x[i] <= lastTower + 4)
            i++
    }

}
```

Running Time

It is clear from the algorithm description and pseudo-code that the running time is,

$$\mathbf{O(n\ log\ n + n)}$$

The $n \log n$ term corresponds to the sorting of the coordinates, which there are n of. In this example we will use merge sort. Merge sort is stable and an algorithm that runs in $n \log n$ in the best, worst, and average case.

The rest of the algorithm corresponds to looping through these n coordinates and doing constant work at each iteration.

Therefore, the overall running time is given by $O(n \log n + n)$, which is the sum of the each respective part of the algorithm (sorting and then dropping towers).

Problem 3B

Assume (towards contradiction) that there exists an optimal algorithm OPT that is more optimal than the greedy algorithm given in Problem 3A, ALG. That is OPT has fewer towers used than ALG.

Let $a[1...f]$ be the location of the f towers in ALG and $b[1...g]$ be the location of the g towers in OPT. Given OPT is more optimal than ALG, $g < f$. Let k be the index where a and b first differ, that is the first occurrence where $a[k] \neq b[k]$.

By algorithm design there must be a house j located at x_j that is covered by the k^{th} tower in a but **not** towers 1...k-1 in a. More specifically, by algorithm design $a[k] = x_j + 4$. Also, since $b[i] = a[i]$ for $i < k$ house j is also not covered by towers 1...k-1 in b. Therefore, in order for OPT to cover all the houses and be a valid solution $b[k]$ must be between x_j and $x_j + 4$ (inclusive). However, since it has been established that $b[k] \neq a[k]$ the following must hold,

$$x_j \leq b[k] < x_j + 4$$

In other words, the coordinate of the k^{th} tower in b must be to the left of the k^{th} tower in a. We can construct an solution OPT' that is identical to OPT expect that the k^{th} tower is at location $a[k]$, not $b[k]$. More formally, if OPT' has the solution array c it is defined as:

$$\begin{cases} c[i] = b[i] = a[i] & i < k \\ c[i] = a[i] & i = k \\ c[i] = b[i] & i > k \end{cases}$$

OPT' is just as good or more optimal than OPT because it covers house j as well as more or at least as much space as OPT to the right of house j.

We can iterate this process until OPT' converges to ALG. Thus, ALG must be just as good or better than the optimal solution OPT. If ALG is just as good as OPT then ALG must also be optimal. If ALG is better than OPT then a contradiction holds and OPT is not optimal. However, in this case the contradiction implies that there is no solution more optimal than ALG, making

ALG optimal. Therefore, in both cases we have proven that ALG is the optimal algorithm.

Problem 3C

To find the smallest value of k all valid values of k can be searched, using Problem 3a as a subroutine to give the minimum number of towers that can be used. In order to properly use Problem 3a as a subroutine it must be modified slightly. The solution to Problem 3a needs to be made more general in order to accommodate a tower range of k, not 4. To do such all instances of 4 needs to be replaced with k. Additionally, the algorithm needs to return the number of towers used. This means the algorithm simply needs to keep track of the towers it drops and return the total number. Lastly, the subroutine will assume the towers are sorted in non-descending order. This is to avoid the redundancy of all subroutines sorting. This subroutine will be referred to as *MinimumNumberTowers(x, k)*. The steps of the *MinimumNumberTowers* algorithm, modified from Problem 3a is given below:

MinimumNumberTowers

1. Find the first instance of a house that is not in range of the currently placed cell towers. Here, *first* means the left-most house (smallest coordinate value). This will correspond to the house that is k miles to the right of the last placed tower. On the first iteration of these steps the first instance of a house that is not in range will correspond to the house at x_0.
2. Place a tower k miles to the right of the house found in Step 1. In other words, if house j is chosen in Step 1, place a tower at $x_j + k$. Record this tower in array a
3. Repeat Step 1 and 2 until all houses have been covered.
4. return $length(a)$.

Finding a valid range for k

Now that we have properly defined the sub-routine *MinimumNumberTowers* which we will leverage, we need to decide the acceptable values for k, in which to search. The length of the road is m and each tower can cover two times k, since each house can be to the left or the right of the tower. Therefore, the entire space [0, m] requires a maximum of $\frac{m}{2t}$ for tower range to be covered. A couple examples to make this revelation more clear is given below.

t	$k_{\max} = \frac{m}{2t}$	Placement to cover $[0, m]$
1	m/2	$m/2$
2	m/4	$m/2, 3m/4$
2	m/6	$m/6, m/2, 5m/6$
		...

We can actually tighten this upper bound constraint slightly. Since we are given the list of coordinates we can replace m with the value $x_n - x_0$. In other words we can constrain the length of the road from m to the distance between the first and last coordinate.

Now that we have established an upper bound we can chose a lower bound for k. This is quite straight forward, since k must be positive and an integer the lower bound for k is $k = 1$.

Search Process

Naively we could search linearly starting at $k = 1$ and increment until we found a value of k that held. This would be constant time in k.

Since we have defined a lower and upper bound for k (above) we can search this solution space in a binary manner. This ensures the problem will be split in half each iteration, leading to logarithmic, rather than linear, time efficiency.

Algorithm

Given the work above we know have the necessary information needed to outline an algorithm.

1. Use merge sort to sort the list of house coordinates x
2. Set $low = 1$, $high = m/2t$
3. Set $middle = (high - low)/2$
4. Get the minimum number of towers needed to cover all of the houses if the tower radius is $middle$ using the *MinimumNumberTowers* subroutine. Call this minimum value $minTowers$.
5. If $minTowers \leq t$ then this radius of k provides a valid solution. Thus, smaller values of k need to be tried. Set $high = middle$. If $minTowers > t$ then this radius of k does **not** provides a valid solution. Thus, larger values of k need to be tried. Set $low = middle$.
6. Iterate through Steps 3-5 until $low = high$.
7. At this point the smalleset possible value of k will be given by the value low. Return low.

To make this algorithm more concrete the pseudo-code is provided below:

```
findSmallestK(t, m, x) {

    x = mergeSort(x)

    low = 1;
    high = m/(2t);
    middle = (high-low)/2 + low;

    while (!endCase(low, middle, high)) {
        smallestPossible = MinimumNumberTowers(x, middle);
        if (smallestPossible <= t) {
            high = middle;
        } else {
            low = middle
        }
        middle = (high-low)/2;
    }

    return low;
}

// returns whether the given low, middle, high values
// correspond to an end case
// an end case is when all values to the left of middle are too small
// and all values including and to the right are large enough
boolean endCase(low, middle, high) {

    return low == high;
}
```

Running time

The running time is given by,

$$\mathbf{O(n\ log(\frac{m}{2t}) + n\ logn)}$$

As explained above the *MinimumNumberTowers* subroutine no longer sorts the list of coordinates it is given. Therefore, its running time is given by $O(n)$. For a further discussion of why this is the case please reference Problem 3a.

The designed algorithm does a binary search over the space $[0, \frac{m}{2t}$. Every time the algorithm iterates it divides this space by two until the space itself reaches a base case (of size < 3). Thus, the algorithm iterates calling the subroutines $\log \frac{m}{2t}$ times. Each time the subroutine is run it requires n work, thus the algorithm has a running time of $n \log \frac{m}{2t}$ (without considering sorting).

In the algorithm description above it is stated that the list of coordinates is sorted at the start of the algorithm. This can be done in $n \log n$ time using mergesort. Adding this running time to that previously calculated yields the total running time of $O(n \log(\frac{m}{2t}) + n \log n)$