Nicholas Strohmeyer
EE 360C SU2022
Nur Touba, Branch Archer

Programming Assignment Report # 3

(a) ***Find Minimum Worst-Case Response Time***

The base cases for this problem are as follows:

(1) k >= n (we have an equal or greater amount of stations than houses)
   (a) Note this also covers the case when we are only looking at a single.house since the way I set up the algorithm we will always allocate at least a single station to a given subproblem (0 stations would be a trivial problem with no solution)
   (b) In this base case, we simply put the stations at the exact same coordinates as the houses. Optimal response time is 0

(2) k = 1 && n > k. This is the case where we only have a single station (and at least 2 houses)
   (a) Here, wel put the station at the midpoint of two houses furthest away from each other. Because we expect the input of our program to be sorted, we can safely use the first and last houses in the array.
   (b) Mid = (pos of first + pos of last) / 2

The recursive equation for the algorithm is :

response = max{ OPT(0, i, k-1) , OPT(i+1, n, 1) } for all i → 0 <= i <= n-1

Where the parameters of OPT are “start pos”, “end pos”, “# of stations.” This recurrence will go through all possible combinations of allocating stations to certain subintervals. The subintervals can then be solved recursively or looked up. We take the maximum of the optimum response time from each subinterval as this will give us the overall “worst-case best response time.”

The memoization rule in my algorithm is**:** if the start position is 0, then remember the solution to the problem. Note that OPT(i+1, n, 1) is always a base case because we always allocate one station to this subproblem. This will make the algorithm vastly more efficient than brute force recursion because every (0, m, k-1) subprobelm (where 0 <= m < n) will only ever have to be solved once, thus eliminating ever repeating any recursive computation that has been done before.

It’s natural to memoize the (0,m) interval because the iterative part of the algorithm will build the solutions bottom-up. That is, starting with the (0,0) solution, then (0,1), then (0,2) etc. So that by the time we see the entire problem (0,n), each (0,m) interval has already been calculated and memoized.

Pseudocode:

Globally Given by problem:

n = number of houses ; **k** = number of stations ; **X** = house positions

Globally Define:

P = n x k matrix where an entry is an array of optimal station positions for (0, t, j)

R = n x k matrix where an entry is optimal response time for subproblem (0, t, j)

public TownPlan **findOptimalResponseTime**(TownPlan **town**) {

1. Get n, k, X from town
2. Intialize **R** → if k > n (put 0) else put "**-1"** : a flag that means "not yet memoized"
3. Intialize **P** with empty ArrayLists []
4. Call **findOptimalTown(** 0 **,** n-1 **,** k) = "**bestTown**"
5. Integer **bestResponse** = **bestTown**.getResponseTime(); (or look up **R**[n-1,k-1])
6. **town**.setResponseTime(**bestResponse**)
7. Return **town;**

}

public TownPlan **findOptimalTown**(int **start**, int **stop**, int **j**) {

1. Set memo flag : (if the **start** = 0 then either it is memoized or can be memoized)
2. *// Initialize some things:*
   - **slice** = **X[start, stop]** *// slice of housePositions array*
   - **m** = size of slice;
   - **currBest** = -1; *// response time variable for this subproblem (-1 flag)*
   - **Stations** = empty Arraylist *// store solution here*
   - **subTown =** create a town object from these subproblem params
2. *// Check if problem is memoized*: (if memoflag = true & **R**[m, k] != -1)
   - If True:
     - Set optimalTown
     - return optimalTown
3. *// Check Base Cases:*
   - **Case 1**: **If** k >= m:
     - for all positions in **slice**
       - **stations.**add**(position)**
       - **currBest** = 0
   - **Case 2: else if** k == 1 && m > k
     - Calculate **midpoint** = **X**.get(**start**) + **X**.get(**stop**) / 2 ;
     - **stations.**add(**midpoint**)
     - **currBest** = **midpoint - X**.get(**start);**
     - *// note: even length intervals or else we'd need to check for a possible difference in length with the midpoint to start/stop*

```
4. ELSE  {  // Recursion :
5.       for i ← 0 to m-1 {
6.         lowerTown = findOptimalTown(start, i , j-1)    // another subproblem
7.         upperTown = findOptimalTown(i+1, stop , 1)  // base case
8.         r = max( lowerTown.getResponseTime(), upperTown.getResposneTIme())
9.         If ( currBest = -1 OR currBest > r ):
10.            currBest = r; // update currBest
11.            Stations = lowerTown.getStations + upperTown.getStations  // merge arrays
         }
   }
12. // Lastly, Memoize & Return Optimal :
13. Solution = setTown(currBest, stations)  // basically just use class set methods
14. If (memoFlag == True) : Memoize (update matrices[m , j] with Solution
14. return Solution
```

(b) ***Find Police Station Positions***

PseudoCode:

```
public TownPlan findOptimalPoliceStationPositions(TownPlan town) {
      1.  Get n, k, X from town
      2.  Intialize R → if k > n (put 0) else put "-1" : a flag that means "not yet memoized"
      3.  Intialize P with empty ArrayLists []
      4.  Call findOptimalTown( 0 , n-1 , k) = "bestTown"
      5.  ArrayList positions = bestTown.getPoliceStationPositions(); (or look up P[n-1,k-1])
      6.  town.setPoliceStationPositions(positions)
      7. Return town;
}
```

(C) ***Runtime Anlaysis:***

I believe this algorithm will run in ***O(n^2)*** time. Here is the logic:

findOptimalTown() has 2 recursive calls: **upperTown** & **lowerTown**. One of these calls is always guaranteed to be a base case, but the other will be a recursive call whose search space depends not only on n but also on k. Because we are memoizing, we should be able to approximate the amount of operations by counting the total # of base cases in the search space, each of which should only need to be calculated once.

Worst case: k = n/2 :

Analysis of my runtimes revealed that the wrorst case should be right when k is about half the size of n. The intuition is “distance from the base cases.” When k is very small, the # of recursive calls is limited until we bottom out with a base case of 1 station. If k is close to n, then we end up with a situation where we have many recursive calls, but only for the first few iterations of the for loop. Soon we will call the interval sizes : (n-k+1, n) with 1 station and (0,k) with k stations, but notice that (0,k) with k stations is our other base case. All further iterations of the loop will be in constant time because we have hit another bottom, this time with our other base case.

Therefore, it makes sense the worst time is when k = n/2 (right in the middle of these two limiting factors). We get the same conclusion by analyzing the runtime. Looking at the for loop, **upperTown** will always be a base case. For lowerTown(), we calculate (0, i, k-1) where (0 <= i < n) and k = # of stations. We would recursively break this interval down k-1 times into k-1 base cases (so long as i >= k). However when i < k, this will be our other base case. So for all intervals (0, i, k) where i < k, we would only calculate 1 base case for lowerTown. This would occur exactly k times in the for loop.

Overall that gives : k * 1 base case(s) (when i < k) and (k-1)*(n-k) base cases (when i > k). Adding that with our upperTown() base cases : **2k + k*(n-k)** total base cases This is dominated by k(n-k), and we can see again why the maximum occurs at k = n/2 since this is when the expression k(n-k) is maximized.

Since the worst case is (n/2)(n/2), we can conclude our algorithm is **O(n^2)**

Below is a screenshot from desmos which shows the parabolic shape of my runtimes :