Homework 2

Authors: Aidan Dalgarno-Platt and Liam Coleman   
  
Problem 1)

a)

* The algorithm makes 3 comparisons, 3 moduli, 3 additions, 2 subtractions, and 1 division for a total of 12 operations
* However, two of those additions only happen separately from one another, and the three moduli only happen when n = 1, so we can roughly subtract 4 operations for a total of 8
* The algorithm then recursively calls itself twice on two arrays of n/2
* T(n) ≤ 8 + T(n/2) + T(n/2)

b)

* T(n) ≤ 8 + T(n/2) + T(n/2)
  + T(n) = 8 + (8 + T(n/4) + T(n/4)) + (8 + T(n/4) + T(n/4))
  + This pattern continues until T(1), which involves 1 comparison and 3 moduli
  + T(1) = 4, therefore
* Ex. n = 16
  + T(16) = 8 + T(8) + T(8)
  + T(16) = 8 + (8 + T(4) + T(4)) + (8 + T(4) + T(4))
  + T(16) = 8 + (8 + (8 + T(2) + T(2)) + (8 + T(2) + T(2))) + (8 + (8 + T(2) + T(2)) + (8 + T(2) + T(2)))
  + T(16) = 8 + (8 + (8 + (8 + T(1) + T(1)) + (8 + T(1) + T(1)) + (8 + (8 + T(1) + T(1)) + (8 + T(1) + T(1)))) + (8 + (8 + (8 + T(1) + T(1)) + (8 + T(1) + T(1))) + (8 + (8 + T(1) + T(1)) + (8 + T(1) + T(1))))
  + T(16) = 8 + (8 + (8 + (8 + T(1) + T(1)) + (8 + T(1) + T(1)) + (8 + (8 + T(1) + T(1)) + (8 + T(1) + T(1)))) + (8 + (8 + (8 + T(1) + T(1)) + (8 + T(1) + T(1))) + (8 + (8 + T(1) + T(1)) + (8 + T(1) + T(1))))
  + T(16) = 15\*8 + 16\*T(1)
  + T(16) = 15\*8 + 16\*4
* T(n) ≤ 8(n - 1) + 4n
* This means T(n) runs in O(n) time

c)

* A = [1,2,4,8,9], n = 5
* WHATDOIDO(0, 4)
* (0 == 4) = false
* m = (0 + 4)/2 = 2
* (llstreak, lrstreak, lmaxstreak) = WHATDOIDO(0, 2)
  + (0 == 2) = false
  + m = (0 + 2)/2 = 1
  + (llstreak, lrstreak, lmaxstreak) = WHATDOIDO(0, 1)
    - (0 == 1) = false
    - m = (0 + 1)/2 = 0
    - (llstreak, lrstreak, lmaxstreak) = WHATDOIDO(0, 0)
      * (0 == 0) = true
      * return (1 % 2, 1 % 2, 1 % 2)
      * return (1, 1, 1)
    - (llstreak, lrstreak, lmaxstreak) = (1, 1, 1)
    - (rlstreak, rrstreak, rmaxstreak) = WHATDOIDO(1, 1)
      * (1 == 1) = true
      * return (2 % 2, 2 % 2, 2 % 2)
      * return (0, 0, 0)
    - (rlstreak, rrstreak, rmaxstreak) = (0, 0, 0)
    - maxstreak = max(1, 0, 1) = 1
    - (1 == (0 - 0 + 1)) = true
    - lstreak = 1 + 0 = 1
    - (0 == (1 - 0)) = false
    - rstreak = 0
    - return (1, 0, 1)
  + (llstreak, lrstreak, lmaxstreak) = (1, 0, 1)
  + (rlstreak, rrstreak, rmaxstreak) = WHATDOIDO(2, 2)
    - (2 == 2) = true
    - return (4 % 2, 4 % 2, 4 % 2)
    - return (0, 0, 0)
  + (rlstreak, rrstreak, rmaxstreak) = (0, 0, 0)
  + maxstreak = max(1, 0, 0) = 1
  + (1 == (1 - 0 + 1)) == false
  + lstreak = 1
  + (0 == (2 - 1)) == false
  + rstreak = 0
  + return (1, 0, 1)
* (llstreak, lrstreak, lmaxstreak) = (1, 0, 1)
* (rlstreak, rrstreak, rmaxstreak) = WHATDOIDO(3, 4)
  + (3 == 4) = false
  + m = (3 + 4) / 2 = 3
  + (llstreak, lrstreak, lmaxstreak) = WHATDOIDO(3, 3)
    - (3 == 3) = true
    - return (8 % 2, 8 % 2, 8 % 2)
    - return (0, 0, 0)
  + (llstreak, lrstreak, lmaxstreak) = (0, 0, 0)
  + (rlstreak, rrstreak, rmaxstreak) = WHATDOIDO(4, 4)
    - (4 == 4) = true
    - return (9 % 2, 9 % 2, 9 % 2)
    - return (1, 1, 1)
  + (rlstreak, rrstreak, rmaxstreak) = (1, 1, 1)
  + maxstreak = max(0, 1, 1) = 1
  + (0 == (3 - 3 + 1)) = false
  + lstreak = 0
  + (1 == (4 - 3)) = true
  + rstreak = 1 + 0 = 1
  + return (0, 1, 1)
* (rlstreak, rrstreak, rmaxstreak) = (0, 1, 1)
* maxstreak = max(1, 1, 0) = 1
* (1 == (2 - 0 + 1)) = false
* lstreak = 1
* (1 == (4 - 2)) = false
* rstreak = 1
* return (1, 1, 1)

d)

* The algorithm counts how many odd numbers are in a row in the array on both the left side and the right side.
* The left-most int in the result (lstreak) represents the longest streak of odd numbers starting at index 0
  + If the array starts with an even number, lstreak is always 0
* The middle int in the result (rstreak) represents the longest streak of odd numbers starting at index (n - 1)
  + If the array ends with an even number, rstreak is always 0
* The right-most int in the result (maxstreak) presents the longest streak of odd numbers anywhere in the array, starting and ending at any index
* Ex. {4, 2, 3, 7, 5, 7} would return (0, 4, 4)
* Ex. {2, 9, 7, 13, 16} would return (0, 0, 3)

Problem 2)

* a) T(n) = 7\*T(n/8) + n
  + a = 7 | b = 8 | f(n) = n
  + logb(a) = log8(7) ~= 0.936
  + n = Ω(n0.936) and f(n/8) <= c\*n for any c >= ⅛
  + So T(n) = Θ(n)
  + This is the third case of the Master theorem
* b) T (n) = 2T (n/4) + √n
  + a = 2, b = 4, f(n)= √n
  + = 0.5
  + Test for n^0.5 against √n
  + √n = √n
  + √n =
  + The second case of the master theorem applies here.
* c) T(n) = 9\*T(n/3) + n2
  + a = 9 | b = 3 | f(n) = n2
  + logb(a) = log3(9) = 2
  + n2 = Θ(n2)
  + So T(n) = Θ(n2 \* log2(n))
  + This is the second case of the Master theorem
* d)
  + a = 4, b = 2, f(n) =
  + = 2
  + Test for n^2 against
  + n^2 >

n log^2 n =

* + T(n) =
  + The first case of the master theorem applies here.

Problem 3)

The first step of our algorithm is splitting the list of children (bullies) into sublists, with the splitting point being at each instance of a -1 (or a child without a lunch). Since no bully ever swaps with a child without a lunch, and no child without a lunch ever takes a lunch from the child ahead of them, the -1 acts as an impassable barrier.

Besides the -1s, the bully problem basically comes down to counting the number of inversions in the list, and therefore the total number of inversions in the list can be calculated by finding the number of inversions in each sublist and summing the results. An efficient way we learned in class to count inversions is by making a small adjustment to merge sort. Whenever there is a smaller number on the left, we know the numbers to its right will be larger, and so we can add a single line incrementing the global number of inversions by the midpoint minus the current left position when this case comes up. Calling this adjustment to merge sort on each sublist nets the total instances of bullying.

The runtime of splitting the list into sublists along the -1s is O(n), as it adds each element to a sublist until it finds a -1, at which point it skips the -1 and starts a new sublist. Every element is visited exactly once. The adjusted merge sort runs in O(nlog(n)) time for each sublist, but the total combined elements from each sublist is less than or equal to the number of elements in the original list, so their total runtimes add up to less than or equal to O(nlog(n)). In any case, if k is the number of sublists/instances of -1, then the runtime is O(k\*nlog(n)) and therefore still O(nlog(n)) since k can be considered to be a constant. This creates a runtime of O(nlog(n) + n), which is overall O(nlog(n)).

Problem 4)

The first step of our algorithm is parsing the input into a dictionary, where the keys are delivery time units and the values are lists of all the expiration dates of the foods delivered in that time unit. Also, the expiration times are added together with the arrival time, since food only starts going off once it has arrived. For instance, the input 3-1 becomes {0: [5, 3, 6], 1: [2, 3], 3: [12]}. The maximum expiration time is also found during this step. Also, every key value (aka every delivery date) is recorded in its own list. Overall, this section of the algorithm runs in O(n) time.

Next, the algorithm runs merge sort on every value in the dictionary, sorting the expiration dates from least to greatest. Each of these merge sorts obviously runs in O(nlog(n)) time. However, the individual sizes of each sublist all add up to the original list, so no more than n values or being sorted by merge still, making this section of the algorithm still O(nlog(n)) in total.

As a final step, the algorithm loops one time unit at a time until the current time unit is greater than the max expiration found in the first step. Within the loop, the algorithm first checks if the current time loop has a delivery on it. If there is, the foods from that delivery are merged with the ongoing list of delivered items (so the foods are still ordered by expiration date from least to greatest after the new food is added).

Next in the loop, the algorithm checks if there are still foods left in the kitchen. If there are no foods left (meaning there was no delivery and there’s no food in the pile), the current time unit jumps to the next delivery date using the list of delivery dates created in the first step of the algorithm. If there are no deliveries left and no more foods left to use, that means the chef succeeded, and a “YES” is returned. Otherwise, the loop continues to the next iteration.

Finally, the first element in the list of delivered foods is removed (which is O(1) time for a linked list). However, if that food’s expiration date minus the current time unit is less than or equal to 0, that means the food has expired and the chef has failed, so a “NO” is returned. Otherwise, the current time unit is incremented by 1.