Kate Lassiter

**For all algorithms that you are asked to "give" or "design", you must do all of the following to get full credit: (a) Describe your algorithm clearly in English. (b) Give pseudocode. (c) Argue correctness, even if you don't give a formal proof and give a convincing argument instead. (d) Give with an explanation the best upper bound that you can for the running time.*

1. ***A. (5 points) Suppose that when doing a Mergesort you recursively break lists into 3 equal sized sublists (or as close to equal as possible) instead of 2. What is the asymptotic running time of this modified Mergesort?***

Like a binary recursion tree that results from regular mergesort, we can solve the recurrence to find the running time for a three-way merge sort:.

The recursion tree starts at the first level $T(n)$ contributing a cost of $c(n)$.

We split this into three branches $T(n/3) + T(n/3) + T(n/3) + O(n) = T(n)$

Each contributing a cost of $c(n/3) + c(n/3) + c(n/3) = cn$.

Each level besides the root contains 3^i nodes each with cost $c(n/3^i)$, with every level having $3^i c(n/3^i) = cn$ cost.

The very bottom level, with n leafs, each contribute cost $c = cn$.

There are $lgn + 1$ levels in the tree with $n = 3$ leaves.

With all $lg_3 + 1$ levels costing cn each, the cost is $cn(lg_3 + 1) = cnlg_3n + cn$.

This gives a final runtime complexity as $O(nlog_3n)$

B. (20 points) Give a (deterministic) algorithm that, on input a set S of n distinct integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x. The worst-case time complexity of your algorithm should be Θ(n log n).

Description: The algorithm works by first sorting the array into ascending order with mergesort. Then, it starts looking at the very first and last elements in the list and comparing. If these two element's sum exceeds the desired value, then shift the right pointer to the next largest item, because if we increment the left pointer by any other number all the way to n, the left value will only grow, causing the sum to grow even higher above the desired sum. If the two element's sum is less than the desired sum, increment the left index by one. This is because if the sum is already too small compared to the desired sum, incrementing the right pointer by one all the way to n will only make the sum smaller because the right pointer moves in descending order. If it finds the sum value it is looking for, the algorithm halts and returns True, stopping the algorithm as soon as possible (i.e as soon as the first pair is found). This allows us to do n comparisons while also checking all the necessary sums.

Pseudocode:

```
A is an Array
sum_value is an integer sum to search for
SumChecker (A,sum_value)
  MergeSort(A,1,len(A)) to sort the list ascendingly
  Initialize the left_index to be 0 and the right_index to be A.length -1
  While left_index<right index do
       Let x, y be the elements pointed to by left_index, right_index
       Compare x,y and if it is below the sum_value
              Advance the left pointer by 1
```

```
            Compare x,y and if it is above the sum_value
                  Advance the right pointer by 1
            Compare x,y and if it is equal to the sum_value
                  Return True
      End while
      Return False
```

Correctness

Base Case: The algorithm correctly returns False for the trivial empty and 1 element array. If there is an empty list there obviously can be no sum to the desired value. For a single element array there are no two numbers that can be summed, so the answer is False, even if that number is the desired sum. If we assume input of 2^n, with n=1 is the base case, the algorithm succeeds because it will sort them ascendingly, then check if the two values sum to the desired value and then return True or False. So this is correct

Loop Variant: We assume input of 2^n, with n=1 in the base case. The 2^{n+1} case is one with 4 elements. In the case of a 4 element set $S = (c_2, c_1, c_4, c_3)$, once S is sorted in ascending order, $S = (c_1, c_2, c_3, c_4)$ we look in S at items $c_1 < c_2 < c_3 < c_4$. If the sum of c_1 and c_4 is the desired sum the algorithm correctly returns True. If c_1+c_4 is greater than the desired sum, we move to c_3 because this is the only way to reduce the sum while only moving one element in the list. If it were to advance to c_2 and $c_1 < c_2, c_2 + c_4 > c_1 + c_4 >$desired value. Thus the pointer is moved correctly in this case. If c_1+c_4 is less than the desired sum, we move to c_2 because $c_2 > c_1, c_2 + c_4 > c_1 + c_4$. This way we are advancing the sum closer to the desired sum. This proves the loop variant approaches the sum or the end of the list.

Termination: If at any point the desired sum is found, the algorithm correctly returns True. If it progresses all the way through the list and never finds the sum, it returns False. Thus proving it is correct.

Running time: Because this algorithm uses merge sort, O(nlogn) is required. After that, just a comparison of the left and right points in the resulting ordered list takes at most n, so it is O(nlogn + n), dropping the lower order term gives an upper bound of O(nlogn).

2. ***(20 points) Consider the following high-level description of a recursive algorithm for sorting. On input a list of n distinct numbers, the algorithm runs in three phases. In the first phase, the first [2n/3] elements of the list are sorted recursively; the recursion bottoms out when the list has size 1 in which case you do nothing, or if the list has size 2, in which case you return the list if it is ordered, otherwise you swap the elements and return the resulting list. In the second phase, the last [2n/3] elements are sorted recursively. Finally, in the third phase, the first [2n/3] elements are sorted recursively again. Give pseudocode for this algorithm, prove its correctness and derive a recurrence for its running time. Use the recurrence to bound its asymptotic running time. Would you use this algorithm in your next application to sort?***

Pseudocode:

```
A is an array
two_thirds_merge (A)
   If A.length greater than 1
         Split array A into the first $2/3^{rd}$ of the list as l1
         list_1=MergeSort(l1,1,l1.length) to sort the list ascendingly
         Append the last remaining $1/3^{rd}$ of the items in A to list_1
```

```
        Create new list l2 by taking the last $2/3^{rd}$ of list_1
        list_2=MergeSort(l2,1,l2.length)
        Append the first $1/3^{rd}$ of the items in list_1 to the front of list_2
        Create new list l3 by taking the first $2/3^{rd}$ of list_2
        A=MergeSort(l3,1,l3.length)
        Append last remaining $1/3^{rd}$ of the data in list_2 to the end of the newly sorted list A
  Return A
```

<u>Correctness:</u>

Base case: When an array is empty or n=1 the loop variant does not begin and returns the trivially sorted empty or single element array, proving this is correct. Assuming n=3 as a base case input, in the first phase the first 2 items of the list are sorted, leaving the last item unsorted. Then in the next phase the last two items are sorted, and the sorted first item is added to the front. Then in the last phase finally the first two items are sorted again. This would return a correctly sorted list so the base case is correct.

Loop variant: Assuming that kn is a multiple of 3, proving correctness can be done on the case of a 6 item list, (k+1)n=2*3=6 items. In the first phase, every item is sorted according to mergesort which has been proven correct in class. So the first 4 items have been correctly sorted ascendingly. Therefore, all the largest items in the first four entries are now at the end of that list and the smallest are at the front. Once we append the two remaining unsorted items from the original list to this sorted list, now we sort the remaining 2/3rd (4 items) in the list, which now contain 2 of the largest elements from the sorting of the first 2/3rd of the list. Because these highest values have been pushed to the middle of the list in the first sorting procedure, and we are taking the remaining 2/3rd which includes these items, this guarantees the largest elements will be pushed to the back of the list. Now the smallest elements in the last 2/3rd of the list are in the middle two positions of the final list. If we perform sorting again on this new list, all the smallest elements in the list are now in the first 2/3rd partition of the list, so mergesort will correctly sort all the elements in the list.

Termination: In the final phase, the first 2/3rd of the list is sorted again, all elements in the list have been sorted and a correctly sorted array is returned.

<u>Running time:</u>

$T(n) = 3T(\frac{n}{2/3}) + c$

By the master method: a = 3, b= 3/2, k=1

$O(n^{log_{3/2}3}) = O(n^{2.71})$

Due to this algorithm running in worse than $O(n^2)$ running time of simple insertion sort, I would not choose to use this algorithm.

3. ***(35 points) You want to measure how similar your musical preferences are to those of your classmates. To this end, you create a list of n songs and you order them as {s1, s2, . . . , sn}, where s1 is your most preferred song and sn your least preferred song. So your ranking for this ordered list of songs is {1, 2, . . . , n}. Now you ask your classmates to also provide a ranking {x1, x2, . . . , xn} for the ordered list {s1, s2, . . . , sn}. Intuitively, a classmate who provides a ranking in almost ascending order has similar musical tastes to you. For example, say you ordered 5 songs in the list {s1, s2, s3, s4, s5}. Your ranking for these songs is {1, 2, 3, 4, 5}. If X ranks the songs as {1, 2, 3, 5, 4} while Y ranks them as {5, 4, 3, 2, 1}, then clearly your preferences are quite similar to X but very different from Y. So one way to measure similarity***

between your preferences and those of your classmates is by checking how far your classmates' rankings are from being in ascending order. In a more abstract setting, you are given a sequence of n distinct numbers x1, . . . , xn and want to understand how far this sequence is from being in ascending order. One way to define a measure for this is by counting how many pairs of numbers xi , xj appear "out of order" in the sequence, that is, xi > xj but xi appears before xj . We will define the disorder of a sequence to be the number of pairs (xi , xj) such that xi > xj but i < j. For example, if the input sequence is {2, 4, 1, 3, 5}, then the pairs (2, 1), (4, 1) and (4, 3) are out of order and the disorder of the sequence is 3.

A. (10 points) Give a brute force algorithm to compute the disorder of an input sequence of size n.

Description: Initialize a number as zero to be the running total of disorder for the algorithm. For every number in the list, check one at a time if each number that comes after that item is less than that current number. If it is, then the numbers are out of order and increase the running total for disorder.

Pseudocode:

```
A is an array
disorder_brute (A)
        Initialize integer variable disorder = 0
        Initialize integer value n = A.length
        if there is at least two elements in A
                for i=1 to n
                   for j=i+1 to n
                        If the A[i] > A[j]
                                Increment disorder by 1
         return disorder
```

Correctness:

Base case: When array is empty or 1 the loop variant does not begin and returns disorder of zero. When input is of size at 2 A={c_1,c_2}, it checks if c_2<c_1, in which case it increases disorder to 1 and returns the correct result.

Loop variant: For every item in the list, it will be compared to every number after it. If it is larger than the one after it, then the array is out of ascending order, and mark that as an increase in the disorder counter. This continues until you get to the end of the list, proving the loop variant is correct in determining the number of items out of ascending order.

Termination: Once it has visited every item in the list and arrives at the last item, it compares it just to itself which will be no disorder, proving the correct number of disordered items has been found.

Running time: Because it loops through the list at most 2 times, the running time is O(n^2)

.

B. (25 points) Can you give a faster algorithm to compute the disorder?

Describe: Utilize mergesort and just add the addition of keeping track of the number of swaps that take place to sort the list. Recursively split the list into two halves, left and right, until either two or a single element remains in each. Then call the process to sort the list with typical mergesort but also calculate disorder as follows. As we progress through the tree, if the second list's current element is less than the first list, this means the rest of the items in list1 are also greater than that number from list2 because the sublists are sorted ascendingly. So if :

List1 {a...c}: $0 < a < c$

List2 {b…d}: $0 < b < d$

If b<a then a…c is also greater than b, meaning that a…c are all out of place, incrementing disorder by the full length of the remainder of list1. If the item in list1 is less than the item in list2, then they are sorted ascendingly and there is no need to increment the disorder counter. Once the algorithm has progressed to the end of one of the two lists, add the rest of the items from the remaining list because they are the maximum values, all sorted, and all greater than those already added to the sorted list. Once this process has completed for both the left and right branches, one final disorder calculation is called on the two branches to count the disorder between them to get to a fully sorted final list. Finally, the disorder calculations from the left and right branches and combined full list are summed to get the final disorder count. This allows for a fast implementation for counting the number of deviations from ascending order utilizing merge sort.

Pseudocode:

```
A is an array
merge_sort_disorder_calc (l1,r1)
    Maintain two pointers, l_index and r_index initialized to point to the first elements in l1
     and r1 respectively.
    Initialize disorder_counter to zero
    Initialize an empty list sorted_list
    while neither index is greater than the length of either list do
        Let x,y be the elements pointed to by l_index,r_index
        Compare x,y and append the smaller to the output sorted_list
        Advance the pointer in the list with the smaller value of x,y
        If y is the minimum of x,y
            Add to disorder_counter the length of left list minus the index of current
             pointer x
    end while
    Add remaining items in l1 and l2 to sorted_list as the are already sorted correctly
    Return disorder_counter, sorted_list

disorder_finder (A)
    If A.length = 1
        return 0 and A
    Split array in 2 halves, l and r
    l_count, l_ordered=disorder_finder(l)
    r_count, r_ordered=disorder_finder(r)
    both_count, A_ordered=merge_sort_disorder_calc(l_ordered,r_ordered)
    all_disorder=sum(l_count,r_count,both_count)
    return all_disorder, A_ordered
```

Correctness: This algorithm performs a simple merge sort, just with the inclusion of counting the number of items that must be switched to get the list to ascending order. Considering it has been proven merge sort is correct, what must be proven is the calculation of the disorder counter.
Base case: If A is a single item list, it will just return 0 disorder and the same array. If we assume n is a multiple of 2, the base case is a two item list, that will be split into single element lists, the lower one placed first in the list, and disorder incremented by 1 if the second number is larger than the first, otherwise zero will be returned. This proves the base case is correct.
Loop variant: The recursion will bottom out when lists are of size 1 or 2, and the process of checking whether the first element in list 2 is less than the current element in list 1 begins. If it isn't, just advance to the next item in the first list. If it is, then that means all the elements in list 1, sorted in ascending order, are also larger than that element in list 2, so the rest of the length of list 1 would all have to be added to the disorder counter because they are all also greater than that number in list 2 as proved in the above algorithm description. This proves the loop variant is correct.
Termination: When all of the sublists have been sorted into two lists, the sum of the disorder from within each list of the two lists is added to the disorder found between the two sorted lists and this is the final total disorder of the list. Thus proving a correct result.
Running time: Because this is just a standard merge sort, and the disorder calculations are in constant time, this algorithm is O(nlogn).

4. ***(35 points) Clustering, that is, grouping together "similar" data points, is an important task in machine learning. The first step for several clustering algorithms is to determine the pair of data points that are closest together (and place the pair in the same cluster). This question examines two algorithms for this step. Closest pair problem Input: a set of n points in the plane {p1 = (x1, y1), p2 = (x2, y2), . . . , pn = (xn, yn)}. Output: the pair (pi , pj) with pi 6= pj for which the euclidean distance between pi and pj is minimized. The euclidean distance d(pi , pj) is given by p (xi − xj) 2 + (yi − yj) 2. For simplicity, assume all xi are distinct, all yi are distinct and n is a power of 2.***
A. (10 points) Give the brute-force algorithm to solve this problem.
Describe: For each point p_i in $p_1 \ldots p_n$ starting at p_i=p_1, compute the distance from this point to every other point in the list up to p_n. Save each point pair combination and each distance as lists. Once distances are calculated from p_i to each point, take the minimum value in this list and this is the minimum distance point to p_i. Save this distance and these points as the minimum distance points. Start the algorithm over at p_{i+1} and continue until p_n. At the end of each iteration, take the running minimum and compare it to the minimum of this iteration. Keep the minimum of these two values.
Pseudocode:

```
A is an array containing $(x_i,y_i)$ points
distance $(X_1,Y_1,X_2,Y_2)$
        xdiff=$(X_1 - X_2)^2$
        ydiff=$(Y_1 - Y_2)^2$
        return $\sqrt{xdiff + ydiff}$

closest pair (A)
        n=A.length
```

```
true_min_distance=NA
pair=NA
**if** n>=2
    **for** i=1 to n
        Initialize empty list distances_i to store distances for the $i^{th}$ index value in A
        Initialize empty list pairs_i  to store points being compared for the $i^{th}$ index value in A
        **for** j=i+1 to n
            Append distance(A[i],A[j]) to distances_i
            Append (A[i],A[j]) to pairs_i
        min_dist=min(distances_i)
        index_of_min= index where distances_i equals min_dist
        min_distance_pair=pairs_i[index_of_min]
        **if** true_min_distance>min_dist
            true_min_distance=min_dist
            pair= min_distance_pair
**return** pair
```

Correctness:

Base case: For the empty list or trivial case of a single element list, the algorithm does not enter the loop variant and NA is returned because there is no minimum distance between a single point and itself or no points. Assume that the length of list A is divisible by 2. For a length 2 list, the distance is calculated between the first pair and the second pair, the distances and pairs are saved. The minimum of a single item list is itself, so the minimum distance is the single stored distance calculation between these two points. Thus the correct result is returned.

Loop variant: For every point in the list, a distance to every other point will be calculated. The minimum of those distances is the minimum possible distance for that point and every other point in the list. If you iterate through every point in the list and complete this same process, the distance between every point and every other point will be calculated. Thus, the minimum of the distances between all these points will be found.

Termination: The algorithm stops when every point has been compared with every other point, when i =n. Thus it stops only when the minimum distance is found and that pair is returned, proving the correct result is returned.

Running time: Because the algorithm iterates through the loop n times, and then loops through all the other n-1 points calculating distances, and the rest of the processes are constant time, the loop takes $O(n^2)$ running time.

B. (25 points) Now consider the following high-level description of a divide-and-conquer algorithm for this problem. • Find a value x for which exactly half the points have xi < x and half have xi > x. On this basis, split the points in two groups, L and R. • Recursively find the closest pair in L and the closest pair in R. Say these pairs are (pL, qL) with pL, qL ∈ L, and (pR, qR) with pR, qR ∈ R, with distances dL and dR respectively. Let d = min{dL, dR}. • It remains to be seen whether there is a point in L and a point in R that are less than distance d apart from each other. To this end, discard all points with xi < x−d or xi > x + d and sort the

remaining points by y coordinate. • Now go through the sorted list and for each point compute its distance to the seven subsequent points in the list. Let (pM, qM) be the closest pair found in this way. • The answer is one of the three pairs (pL, qL),(pR, qR),(pM, qM), whichever has the smallest euclidean distance.

i. (15 points) Give the recurrence for the running time of this algorithm; explain carefully your derivation.

The first operation to be performed is to sort the x_i..x_n value pairs in order to split the list into two halves, one with all points less than some value X and one with all points greater than X. Assume merge sort is used for this process, adding

$$T(n) = 2T(n/2) + O(n)$$

to the recurrence.

Then divide this sorted array into two halves. This contributes constant O(1) time

Then bottom out the recursive processes when lists are of size 1 or two, and calculate the distance for each of these points which is cn and find the minimum. This adds

$$T(n) = 2T(n/2)$$

to the recurrence and returns the minimum distance calculation for both the left and right list.

Taking the minimum of these 2 contributes constant time.

Then, discard all points x_i..x_n that have values less than the median value minus the minimum distance from the two halves or greater than the median plus the minimum distance between the halves. This will contribute

$$T(n) = O(n)$$

to the recurrence.

Then the remaining coordinates have to be sorted by y, which again using merge sort contributes

$$T(n) = 2T(n/2) + O(n)$$

to the recurrence.

Now to find the smallest distance between these coordinates, only 7 points are checked for each n. This contributes

$$T(n) = O(1)$$

to the recurrence.

Then take the minimum of these three smallest distances, done in constant time O(1).

So the recurrence is:

$$T(n) = [2T(n/2) + O(n)] + [2T(n/2)] + [O(n)] + [2T(n/2) + O(n)] + [O(1)]$$

ii. (10 points) Solve the recurrence

By the master method: a=2, b=2, k=1

$$T(n) = [O(nlogn)] + [O(logn)] + O(n) + [O(nlogn)] + O(1)$$

Dropping the lower order terms results in

$$T(n) = [O(nlogn)] + [O(logn)] + [O(nlogn)]$$

$$T(n) = O((nlogn * logn * nlogn))$$

$$T(n) = O((nlogn * n(logn)^2))$$

$$T(n) = O((n^2(logn)^3))$$