CS41 Homework 5

This homework is due at 11:59PM on Sunday, November 22. Write your solution using LaTeX. Submit this homework in a file named `hw5.tex` using **github**. This is a **10 point assignment**.

It's ok to discuss approaches at a high level. In fact, I encourage you to discuss general strategies. However, you should not reveal specific details of a solution, nor should you show your written solution to anyone else.

The main **learning goals** of this homework are to get practice with divide-and-conquer algorithms, as well as dynamic programming.

1. **Closest Pair of Points.** In class, we looked at a divide-and-conquer algorithm for the *closest pair of points* problem, which is the problem of finding the smallest distance between any two points (in two dimensions) in a given list P of n points.

   For reference, the pseudocode is provided below. Most of our interest with the closest pair of points problem is in the "conquer" part, for which the bulk of the work happens in the subroutine ClosestSplitPair. In the below, $d(p, q)$ represents the distance between two points p and q, and the lists P_x and P_y are the points sorted by x and y coordinate, respectively (we precompute these from P before the first call to ClosestPair).

   ClosestPair(P_x, P_y)

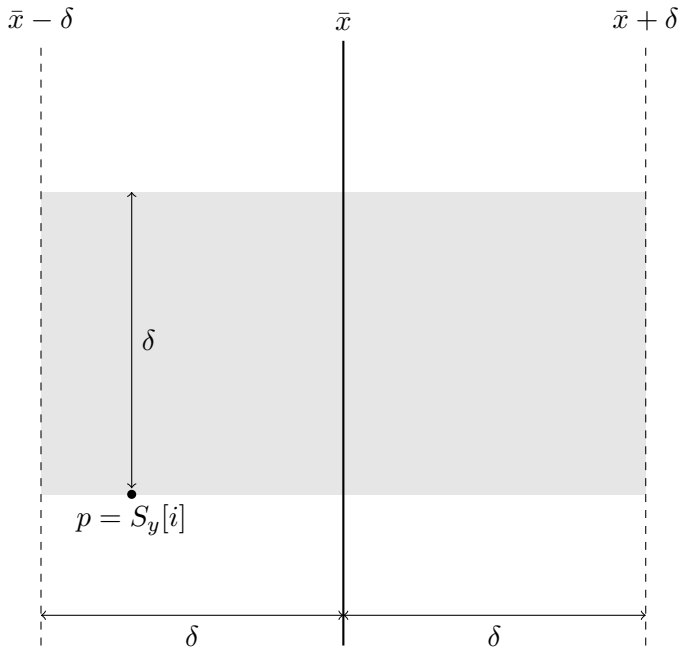
   ```
   if |P_x| ≤ 3:
       compute the closest pair by brute force and return it
   Let L_x be the left half of points, sorted by x-coordinate
   Let L_y be the left half of points, sorted by y-coordinate
   Let R_x be the right half of points, sorted by x-coordinate
   Let R_y be the right half of points, sorted by y-coordinate
   (p_L, q_L) = ClosestPair(L_x, L_y) // Find the closest pair in the left half
   (p_R, q_R) = ClosestPair(R_x, R_y) // Find the closest pair in the right half
   Let δ = min{d(p_L, q_L), d(p_R, q_R)} // δ is the smaller of the two distances
   Let (p_S, q_S) = ClosestSplitPair(P_x, P_y, δ)
   return the closest pair among the three options (p_L, q_L), (p_R, q_R), and (p_S, q_S)
   ```

   ClosestSplitPair(P_x, P_y, δ)

   ```
   Let x̄ be the x-coordinate of P_x[n/2], where n = |P_x|
   Let S_y be the points of P_y (in sorted order) with x value between x̄ − δ and x̄ + δ
   Let mindist = δ and closestpair = null
   for i = 1, 2, ..., |S_y|:
       Let p = S_y[i]
       for j = i + 1, i + 2, ..., |S_y|:
           Let q = S_y[j]
           if the y coordinates of p and q differ by more than δ
               break
           if d(p, q) < mindist:
               mindist = d(p, q) // This pair is the new closest
               closestpair = (p, q)
   return closestpair
   ```

In ClosestSplitPair, $\bar{x}$ is the *median* x value, so that all points in the left half have x-coordinate at most $\bar{x}$ and all points in the right half have x-coordinate at least $\bar{x}$. The ClosestPair function, on line 11, then returns the best of the three options (ClosestSplitPair may return `null` if no pair closer than δ was found; in this case, we just ignore the null pair and return the best of the two other pairs).

In class, we claimed that any split pair closer than δ must have both points in the vertical band lying between $\bar{x}-\delta$ and $\bar{x}+\delta$, which is why we restrict our search to the list S_y constructed in line 2 of ClosestSplitPair. We also argued that such a split pair (with distance less than δ) must have y coordinates differing by at most δ, which justifies the early exit, at line 8–9, from the for loop at line 6. The diagram below shows this (the only points which can be less than δ away from point p must lie in the grey region).



In this part, we will show that there can be at most a constant number of points in the grey region, and thus the loop at lines 6–12 runs for a constant number of iterations for each point $p \in S_y$.

(a) Consider a $\frac{\delta}{2} \times \frac{\delta}{2}$ square that lies entirely on one side of the $\bar{x}$ line (i.e., either on the left half, or the right half). Show that such a square on the left side contains at most one point from the left half, and a square on the right side contains at most one point from the right half.

(**Hint:** Use the fact that δ is defined so that the closest pair of points in the left half, and the closest pair of points in the right half, have distance at least δ).

(b) Consider the grey region in the above diagram. Using the previous part, show that for any point $p = (x_p, y_p) \in S_y$, there are at most *seven* other points in the grey region (that is, the $2\delta \times \delta$ rectangle defined by $\bar{x} - \delta \leq x \leq \bar{x} + \delta$ and $y_p \leq y \leq y_p + \delta$).

(**Hint:** Divide the region into eight $\frac{\delta}{2} \times \frac{\delta}{2}$ squares, four on either side of the $\bar{x}$ line. What does the previous part tell us?)

(c) Conclude that the loop in lines 6–12 of ClosestSplitPair terminates after at most 7 iterations, and that ClosestSplitPair runs in time $O(n)$.

(d) Suppose that in ClosestPair, the sets L_y and R_y are constructed by simply sorting (using merge-sort) the left-half points and right-half points by y-coordinate in every recursive call, so that it takes time $\Theta(n \log n)$ at each call. Write the recurrence relation describing the running time of ClosestPair in this case, and solve it (using a recursion tree or the substitution method) to give an asymptotic running time for this implementation of ClosestPair.

(e) In class, we claimed that we don't actually need to sort from scratch at each recursive call, so that once we have P_x and P_y, constructing L_y, R_y, and S_y can be done in $O(n)$ time. Briefly argue how this can be done, and conclude that ClosestPair (and ClosestSplitPair) can be implemented so that the total running time of ClosestPair is $O(n \log n)$.

(The recurrence relation you get should look similar to one we've seen in class; you can cite its asymptotic solution without re-solving it.)

2. **Quicksort recurrences.** Recall from CS35 the *partition* process used in *quicksort*, given as pseudocode below. Recall also that this procedure runs in time $O(n)$.

The notation $A[1 \ldots n]$ means an array $(A[1], A[2], A[3], \ldots, A[n])$, i.e., an array A consisting of values indexed from 1 through n. For simplicity, we assume that all values in A are distinct.

```
Partition(A[1...n])
 1  Let p = A[n] // p is the "pivot" value
 2  Let i = 1 and j = n − 1
 3  while i ≤ j
 4      if A[i] ≤ p
 5          i = i + 1
 6      elseif A[j] ≥ p
 7          j = j − 1
 8      else
 9          Swap A[i] and A[j]
10  Swap A[i] and A[n] // Places pivot value in its proper position
11  return i // Returns the index of the pivot value
```

Note that after calling Partition on an array, the index i returned is such that everything in the array before index i is less than $A[i]$ and everything after index i is greater than $A[i]$. We can think of A as having been split (or *partitioned*) into a *smaller* portion $A_< = A[1 \ldots i-1]$ and a *larger* portion $A_> = A[i+1 \ldots n]$, with $A[i]$ sitting in between the two.

(a) In quicksort, we first call Partition on an array A, then recursively quicksort $A[1 \ldots i-1]$ and $A[i+1 \ldots n]$. Pseudocode is below for reference. In the *best case*, the pivot value in Partition is always the *median* value of A, so Partition creates a perfectly even split

every time. Suppose we run quicksort and every call to PARTITION creates a perfectly even split (i.e., called on an array of n elements, it returns $n/2$).

Give a recurrence relation that describes the total running time for quicksort in this case, and state the asymptotic running time. You must justify your asymptotic solution, but you may cite the solution to any recurrence relation we've seen in class.

(b) In the *worst case*, PARTITION may create a bad split. Suppose we run quicksort on a list that is already sorted, so the pivot value $A[n]$ is the largest value in the array. Give a recurrence relation that describes the running time in this case, and solve it (asymptotically) using the method of your choice to give the asymptotic running time.

Quicksort Pseudocode for reference. Note that $A[1 \dots q-1]$ means the subarray of A from indices 1 through $q-1$, i.e. $(A[1], A[2], \dots, A[q-1])$. Similarly, $A[q+1 \dots n]$ means the subarray starting at index $q+1$, i.e., $(A[q+1], A[q+2], \dots, A[n])$.

```
QUICKSORT(A[1...n])
1  if n ≤ 1
2      return
3  Let q = PARTITION(A)
4  QUICKSORT(A[1...q-1])
5  QUICKSORT(A[q+1...n])
```

3. **Finding the kth Smallest Number.** Here we will use the PARTITION subroutine to write a divide-and-conquer algorithm to find the kth smallest number in an array.

(a) Consider the incomplete pseudocode below.

```
KTHSMALLEST(A[1...n], k)
1  if n == 1
2      return A[1]
3  q = PARTITION(A) // A[q] is the qth smallest value in A
4  // TODO: Find the kth smallest using divide-and-conquer
```

As noted in the previous problem, PARTITION rearranges the array and returns a value q such that all values before index q are smaller than $A[q]$ and all values after index q are larger than $A[q]$.

Use this fact to finish the pseudocode to give a divide-and-conquer algorithm for finding the kth smallest value in the array A.

NB: Any single call to KTHSMALLEST should make one recursive call (see the next part).

(**Hint:** Consider how q compares to k. What can you conclude when $k = q$? When $k < q$? When $k > q$?)

(b) Suppose every time your algorithm calls PARTITION, the pivot value is the *median* value of the array (the best-case scenario), so that PARTITION splits the array perfectly in half, and that your algorithm continues until it reaches the $n = 1$ base case. Write a recurrence relation to describe the running time of your algorithm in this case. Use the method of your choice to find the asymptotic running time from your recurrence relation.

(c) Suppose instead that we have the *worst case* situation, where A is sorted and PARTITION chooses the largest value as the pivot. Suppose we run your algorithm with $k = 1$ in this case. Write a recurrence relation that describes the running time in this case, and compare your recurrence to the one from problem 2(b).

4. **Even Split.** You and a friend have acquired a collection of goodies with different values. You have n items whose values are $a_1, a_2, \ldots, a_n$. You would like to figure out if you can divide the items evenly amongst yourselves, by splitting them into two groups so that the total value of each group is equal. Our goal is to devise an algorithm that will determine whether this is possible.

   As an example, given items with values $3, 1, 2, 1, 1, 2$, we *can* find an even split: $3 + 1 + 1 = 5$ and $2 + 2 + 1 = 5$. On the other hand, given items with values $1, 3, 6$, we *cannot* split them evenly.

   (a) Let $\text{opt}(i, T)$ be `True` if we can find a subset of the items $a_1, a_2, \ldots, a_i$ whose total value is T. Give a recursive definition for $\text{opt}(i, T)$, and briefly explain why it is correct (e.g., why it covers all possible cases). Don't forget to include base cases.

   (b) In what order do the entries of a table for $\text{opt}(i, T)$ need to be filled in? Briefly explain your answer.

   (c) Using your answers to the previous two parts, give pseudocode for a dynamic programming algorithm to solve the *even split* problem. Justify that your algorithm runs in $O(nS)$ time, where $S = \sum_{i=1}^{n} a_i$ is the sum of all the values.

   (d) Assume that your algorithm receives the numbers represented in *binary*. Is this algorithm's running time polynomial in the *input size*. Justify your answer.

   (**NB:** Consider how large each value a_i may be when written using b bits, and how large the sum of n such b-bit numbers might be; how does S relate to the input size?)