CSC-421 Applied Algorithms and Structures
Spring 2017

Instructor: Iyad Kanj
Office: CDM 832
Phone: (312) 362-5558
Email: `ikanj@cs.depaul.edu`
Office Hours: Monday 4:40 - 5:40 PM & Wednesday 4:45 - 6:45 PM
Course Website: https://d2l.depaul.edu/

Solution Key to Assignment #2

1. I will skip this one because we did a similar example in class.

2. I will skip this one because we did a similar example in class.

3. The idea is to sort the points by their x-coordinate first, and whenever two points have the same x-coordinate, to sort them by their y-coordinate. After sorting the points as described, identical points must appear contiguously/adjacently in the sorted array; so now we can scan the array, checking if any two adjacent points are identical, which can be done in $O(n)$ time. We describe how to sort the points as described above in $O(n \lg n)$ time. To do so, we slightly tweak any $O(n \lg n)$-time sorting algorithm, say **Merge Sort** for example, as follows. In the procedure **Merge** (in **Merge Sort**), whenever two elements $A[i]$ and $A[j]$ are compared via the comparison "$A[i] \leq A[j]$", we replace the comparison with the following. (Note that $A[i]$ and $A[j]$ contain points.) Suppose $A[i]$ contains point $p_i(x_i, y_i)$ and $A[j]$ contains point $p_j(x_j, y_j)$. We replace "$A[i] \leq A[j]$" with: "$(x_i < x_j)$ or $(x_i = x_j)$ and $(y_i \leq y_j)$". The running time of **Merge**, and hence, of **Merge Sort**, remains the same because the modified comparison still takes constant time. The overall running time of the algorithm is $O(n \lg n)$ (sorting) plus $O(n)$ (scanning the array afterwards), which is $O(n \lg n)$.

4. For each point p_i in the set (this iterates n times): sort the remaining points (p_j's, $1 \leq j \leq n$ and $j \neq i$) by their polar angles with respect

to p_i (in $O(n \lg n)$ time) and check for duplicates ($O(n)$ time); if duplicates exist return TRUE. If after going through all points we do not find duplicates, we return FALSE. The running time is $O(n^2 \lg n)$.

5. The recursive procedure is given next.

```
Insertion-Sort(A, 1, i)

if i > 1 then
  Insertion-Sort(A, 1, i-1);
key <-- A[i]; j <-- i-1;
while (key < A[j]) and (j > 0) do
  A[j+1] <-- A[j];
  j <-- j-1;
A[j+1] <-- key;
```

To analyze the worst-case running time of the algorithm, observe that when there is only one element in the array we do no comparisons. When we have n elements, we call the algorithm recursively on $n-1$ elements and then we insert the last element. Inserting the n-th element can take in the worst case $n-1$ comparisons. We can express the worst-case running time of the algorithm by the recurrence relation $T(1) = 0$ and $T(n) = T(n-1)+n-1$ for $n > 1$. You have seen that this recurrence has solution $T(n) = \Theta(n^2)$ (recurrence relation for Quick Sort).

6. (a) Each sublist has k elements and can be sorted in time $\Theta(k^2)$ using Insertion-Sort. Since we have n/k lists, all the lists can be sorted in time $(n/k)\Theta(k^2) = \Theta(nk)$.

   (b) We do the merging in levels. At the bottom level, level 0, we have n/k sublists, each of size k. We group the sublists into groups each containing two sublists, and we merge them using the same merge subroutine that we used in Merge-Sort. Merging two sublists takes time linear in the total number of elements in the two sublists. The total time taken by the algorithm at the bottom level is $O(n)$. The number of sublists to be merged at level 1 becomes $n/2k$, we again apply the same strategy grouping the sublists into groups each containing two sublists and we merge them. The time spent to merge the sublists at level 1 is also $O(n)$. We keep repeating this step until we have one whole list. Since at each level the number of sublists is reduced by half, it will take us $\lg{(n/k)}$ levels to reach the top level where we have only one list. At each level the time spent is $O(n)$. The total running time is $O(n\lg{(n/k)})$.

(c) $k = \Theta(\lg n)$.

7. (a) To argue that the line exists, pick a point p_0 with the smallest y-coordinate. Assume, without loss of generality, that p_0 corresponds to a buster; the argument is the same if the point is a ghost. Sort the remaining points p_j's in an increasing order of their polar angle with respect to p_0; let the sorted list be $p_0, \ldots, p_{2n-1}$. Consider a line L that passes through p_0 and such that all the remaining points are on one side of L. This is possible because p_0 is the lowest point in the set, and no three points in the set are collinear. Start rotating line L anticlockwise around p_0 until you hit the first point p_1 (p_1 might be on (L) to begin with). If p_1 is a ghost then we are done, the line p_0p_1 is the desired line. Otherwise, p_1 is a buster. Keep rotating L. Just as L leaves point p_1, the number of busters in the half plane determined by L and containing p_1 is strictly larger than the number of ghosts. When L reaches the last point in the set, the number of busters in the half plane determined by L and containing p_1 is equal to that of the ghosts. Therefore, there must exist a first point p_i (smallest index i) such that when L reaches p_i the number of busters in the half plane determined by L and containing p_1 is equal to that of the ghosts (this point could be p_{2n-1}). Since originally the number of busters in the half plane determined by L and containing p_1 was more than that of the ghosts, and since p_i is the first point such that when L hits p_i the two numbers become equal, p_i must be a ghost. The line p_0p_i is the desired line.

The above procedure is constructive; it describes how line p_0p_i can be computed. This involves finding the point p_0 with minimum y-coordinate ($O(n)$ time), sorting the remaining points around p_0 ($O(n \lg n)$ time), and scanning the list of points in the sorted order while keeping track of the number of ghosts and busters below L after each scanned point ($O(n)$ time). Therefore, finding such a line can be done in $O(n \lg n)$ time.

(b) The algorithm starts by finding a line passing through a ghost g and a buster b satisfying the properties described in (a); this step can be done in $O(n \lg n)$ time. The ghost g is paired with the buster b, and both are removed. The algorithm is called recursively on either side of this line. The worst case happens when one side of the line is empty and gives the recurrence $T(n) = T(n-1) + O(n \lg n)$ (n is the number of pairs). Using the iteration method, we get $T(n) = O(n^2 \lg n)$.