Brandon Vo

Homework 7

1. 7.4.1 from CLRS on page 184

Show that in the recurrence

$$T(n) = \max_{0 \le q \le n-1} \big(T(q) + T(n-q-1)\big) + \theta(n)$$

$$T(n) = \Omega(\mathrm{n}^2)$$

- Page 180-181: maximum bound of T(q) + T(n-q-1) is positive with respect to q. This also means that the bound of (q^2 + (n – q – 1)^2 ≤ (n-1)^2 is n^2 – 2n + 1
- So we can substitute every value of q to be (n-1)

$$IH\!:\ T(n) \ge dn^2$$

$$T(n) = \max_{0 \le q \le n-1} \big(T(q) + T(n-q-1)\big) + \theta(n)$$

$$\ge d(n^2 - 2n + 1) + \theta(n)$$

$$\ge dn^2 - d(2n - 1) + \theta(n)$$

$$\ge dn^2 - 2dn + d + cn$$

$$\ge dn^2\ if(-2dn + d + cn) > 0\ and\ c > 1$$

Brandon Vo

2. Question 7-1a on page 185 in the textbook.1

```
HOARE-PARTITION(A, p, r)
x = A[p]
i = p - 1
j = r + 1
while TRUE
    repeat
        j = j - 1
    until A[j] ≤ x
    repeat
        i = i + 1
    until A[i] ≥ x
    if i < j
        exchange A[i] with A[j]
    else return j
```

a. Demonstrate the operation of HOARE-PARTITION on the array A = (13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21) showing the values of the array and auxiliary values after each iteration of the while loop in lines 4–13.

- Before the loop:
  - I = 1, A[i] = 13
  - J = 12, A[j] = 21
  - ⟨13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21⟩
- First iteration:
  - X = 13
  - A[i] = 13, I = 1
  - A[j] = 6, j = 11
  - X = 13 where A[j] ≤ X and A[i] ≥ X
  - 6 ≤ 13 and 13 ≤ 13
  - i < j, swap 13 and 6
  - ⟨6, 19, 9, 5, 12, 8, 7, 4, 11, 2, 13, 21⟩
- Second iteration:
  - X = 13
  - A[i] = 19, I = 2

- A[j] = 2, j = 10
- X = 13 where A[j] ≤ X and A[i] ≥ X
- I < J, so swap 19 and 2
- ⟨6, 2, 9, 5, 12, 8, 7, 4, 11, 19, 13, 21⟩

- Third iteration:
  - X = 13
  - A[j] = 11, j = 8
  - A[i] = 19, i = 9
  - J < i, so returning j
  - Hoare's partition ends at ⟨6, 2, 9, 5, 12, 8, 7, 4, 11, 19, 13, 21⟩

Brandon Vo

3. If the array contained many duplicate items, which would be a better partitioning algorithm: Hoare, or the one presented in class?

Reference: https://cs.stackexchange.com/questions/11458/quicksort-partitioning-hoare-vs-lomuto

Quicksort for both cases risk the worst-case time being $O(n^2)$. Whenever quicksort has a bad pivot, the time complexity slowly degrades to the worst case.

Quick sort using a random pivot runs a risk where if a majority of the elements in an array are duplicates, then the pivot will have a high chance of randomly setting one of the duplicate elements as its pivot. If chosen, it will see $A[j] \leq x$ and swap for every other duplicate element in the array.

- If we have an array of all duplicate elements, then randomized quicksort would always have a worst-case pivot when randomly selecting. Whenever a pivot is chosen, it will swap with every element in the array. This means in an array with all duplicates, every element swaps with every other element, costing $O(n^2)$ time.

In Hoare's partitioning algorithm, because we check for less than or equal to and greater than or equal to for i and j, any duplicate elements found will all be discovered by the pivot and be placed in one of the two sub-arrays. This means that for an array with mostly duplicate elements, i and j will always swap when they get a hold of two duplicate elements.

However, Hoare's partitioning algorithm always have I and j move closer towards each other with each swap. The greater number of duplicate elements would guarantee a swap, but it also increases the likelihood of i and j intersecting their pivots in the middle of the array as they both move towards each other.

- In an array with all duplicate elements, Hoare's partition will have i and j swap until they meet in the middle, making this $O(n \log n)$ time.

Brandon Vo

4. Professor Williams (having looked at the proof on page 1041) comes up with a scheme that allows the closest-pair algorithm to check only 5 points following each point in array Y'. The idea is always to place points on line *l* into set P_L. Then, there cannot be pairs of coincident points on line *l* with one point in PL and one in P_R. Thus, at most 6 points can reside in the $\delta \times 2\delta$ rectangle. What is the flaw in the professor's scheme?

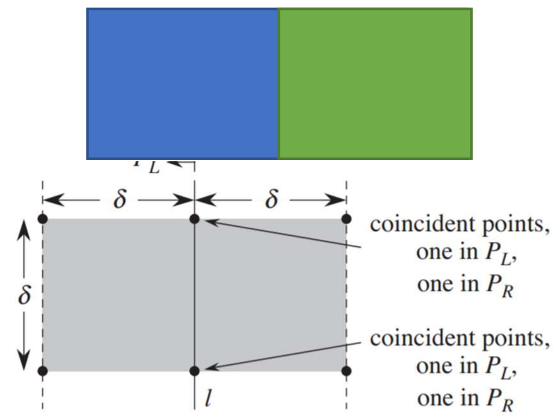
Professor Williams assumes that we can simply compare all points from P_L to P_R faster using his method. However, this does not take into account a scenario where all points are in P_L and no points are in P_R or a scenario where all points found share the same x-coordinate. It also does not take into account the possibility of an uneven number of points in P_L and P_R.

The combine part of the algorithm needs to check the pairs of points between P_L and P_R and determine if there is a pair whose distance is less than δ. If we use Professor William's method and there are no points in P_R, then we will fail to merge P_L and P_R because William's extra points are searching for a point in P_R that does not exist.

Brandon Vo

5. CLRS question 33.4-2 on page 1043

Show that it actually suffices to check only the points in the 5 array positions following each point in the array Y'.



(b)

When judging distances, the maximum number of coincident points we can place would be at each corner of the rectangles. When compare based on a rectangle of size δ x 2 δ, there are 6 coincident points we can place where there would be no comparison overlap.

- We're concerned if two points have a distance < δ
- Then, if we're presented with two squares of size δ x δ, the points we would check would avoid overlap by being δ distance from each other
    - This δ distance would mean each point is placed at each corner of each square

If we place one point at one of the 6 points, then we're left with the other 5 coincident points to check for the smallest distance. Therefore, we only need to check at most 5 other points which don't overlap in a δ x δ square.

Brandon Vo

6. CLRS question 33.4-6 on page 1044

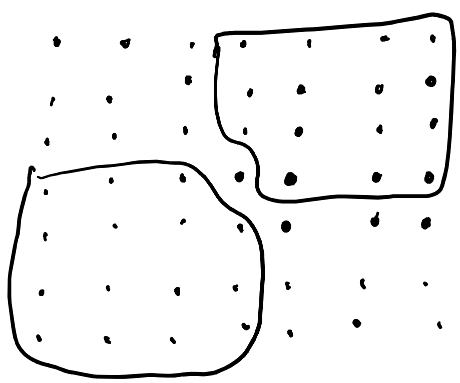
Suggest a change to the closest-pair algorithm that avoids presorting the Y array but leaves the running time as O(n lg n). (Hint: Merge sorted arrays Y_L and Y_R to form the sorted array Y.)

The main algorithm already splits the distance Y into Y_L and Y_R. recursively, we can put them back together in O(n log n) time. We apply a merge step on Y_L and Y_R and have them merge back into Y. While merging, we would return the points found based on the order of their Y-coordinates. Since merging two arrays is done in O(n) time, the overall runtime is not affected and will remain to be $O(n\ log\ n)$.

7. In the O(n) worst case deterministic select algorithm, the pivot was found by dividing the input elements into groups of 5 and then finding the median of their medians. Would the algorithm still run in O(n) time if we divided the elements into groups of 7 instead of 5? Would the algorithm still run in O(n) time if we divided the elements into groups of 3 instead of 5? Prove your answer. ∗Many of these questions came from outside sources. [1]

N elements will be divided into groups of size 7. Our # of median elements < x would be n/7. Because it takes O(1) to fine a median in a pool of constant numbers, we can find our pool of median representatives in n/7 * O(1) time which is still O(n) time.

- Time to partition into 7 groups: O(n) time
- Time to group and find the median representatives: T(n/7)
- The time to check the elements that weren't discarded after checking the median representatives: $\left\lceil \frac{7}{2} \right\rceil = 4\ groups\ to\ check\ after\ searching\ the\ medians$
    - $T\left(\frac{14-4}{14} * n\right) = T\left(\frac{10n}{14}\right)\ elements\ to\ check\ from\ the\ remaining\ partitions$



$\frac{n}{7}$ medians partitions

$\left\lceil \frac{7}{2} \right\rceil = \frac{4n}{7}$ medians to check

$\frac{10n}{14}$ remaining elements to check

$$T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{5n}{7}\right) + \theta(n)$$

$$IH: T(n) \le dn$$

$$T(n) \le d * \frac{n}{7} + d * \frac{5n}{7} + cn$$

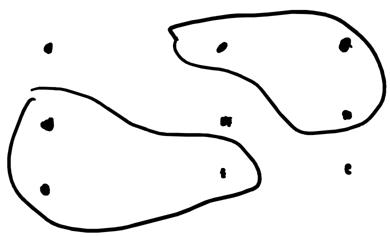
$$T(n) \le \frac{6d}{7} + cn$$

$$\le O(n)\ if\ c < \frac{1}{7}$$

This would cost less than O(n) only if our constant time is less than 1/7.

Brandon Vo

If we divided it into groups of size 3, it would take n/3 * O(n) time to search for an element and group the elements into partitions based on their medians. It would also cost T(2n/3) time to check the upper/lower elements and discard them during comparisons after the median was found.



$T\left(\frac{n}{3}\right)$ for groups of 3

$\left\lceil \frac{3}{2} \right\rceil = \frac{2n}{3}$ medians to check

$\frac{6-2}{6} = \frac{4n}{6}$ elements after discarding

- Time to partition into groups of size 3 would be O(n) time
- Time to select the median of medians would be T(n/3)
- After selecting the median of medians, we would have to check $\left\lceil \frac{3}{2} \right\rceil = 2$ medians to see which group to search through
    - This leaves us with $T\left(\frac{6-2}{6}n\right) = T\left(\frac{4n}{6}\right)$ *elements to search through*

$$T(n) = \theta(n) + T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right)$$

$$IH\ checking\ for\ \Omega\colon\ T(n) \geq\ dn$$

$$T(n) \geq cn + d * \frac{n}{3} + d * \frac{2n}{3}$$

$$\geq cn + dn$$

$$Using\ induction, when\ handling\ groups\ of\ size\ 3, our\ selection\ algorithm\ is\ \Omega(n)$$

Brandon Vo

8. Suppose University X (UX) is electing its student-body president. Suppose further that everyone at UX is a candidate and voters write down the student number of the person they are voting for, rather than checking a box. Let A be an array containing n such votes, that is, student numbers for candidates receiving votes, listed in no particular order.

• Your job is to determine if one of the candidates got a majority of the votes, that is, more than n/2 votes. Describe an O(n) worst case time algorithm for determining if there is a student number that appears more than n/2 times in A.

• Consider the election problem from the previous exercise, but now describe an algorithm running in O(n) worst case running time to determine the student numbers of every candidate that received more than n/3 votes. This question is modified from questions in Goodrich, Michael T.; Tamassia, Roberto. Algorithm Design and Applications

- If majority means > n/2, then we can use deterministic selection to find the list
- If an element is in the majority, then it would be counted the most when iterating over the array
- So we could just go over the list and count the element in our array. If we have a majority element, it would be the last element

1	2	4	1	1	1	2

For the array above, the number of occurrences for each element would be

- #1: 4 occurrences
- #2: 2 occurrences
- #4: 1 occurrence
- We can create a hash map which will be used to keep track of what candidates we have based on their collisions
  - This hash map will also include satellite data counting every collision or vote we received for that candidate
  - Every collision found will increase the counter for that candidate's votes
- After counting all votes, we can go over the hash map list and check for the candidate who has > n/2 votes
- This process will take O(2n) if every element is unique because then we would end up with a hash map of size m = n and find out that there is no majority winner.
- We only need to traverse the list once and the hash map once, making this O(n)

ELECTION(A)

//Where array A contains the student votes

Create Hash Table LIST that has counter as satellite data

// θ(n) time

For i = 1 to A.length

Brandon Vo

If HASH(A[i]) does not have a collision with LIST

//Assuming no unintentional collisions between two different candidates

Hash-Insert HASH(A[i]) into LIST

LIST(H(A[i]).counter = 1 //Add satellite data for the counter

List(H(A[i]).index = i

Else //Collision found

LIST(H(A[i]).counter++) //Increment the counter

EndFor

//Go over the hash table we have and count the collisions

$//\theta(k)$ *time where* $k \leq n$

For (i = 1 to LIST.length)

If(LIST[i].counter > n/2)

Print A[List[i].index] //Print the candidate

Return

EndFor

//If we have reached this point and not returned, that means there was no majority candidate found

Print "There is no majority candidate"

- To find the candidates with over n/3 votes, our hash map already creates a list of how many votes each candidate receives.
- We can simply go over the hash map and look for any candidate that has > n/3 votes.
- This still runs in O(2n) which is still O(n) because we're traversing the array and hash map once.

The function is the same as before except we check the hash table for any candidate with n/3 collisions

//Where array A contains the student votes

Create Hash Table LIST that has counter as satellite data

For i = 1 to A.length

If HASH(A[i]) does not have a collision with LIST

```
        Hash-Insert HASH(A[i]) into LIST
        LIST(H(A[i]).counter = 1  //Add satellite data for the counter
        List(H(A[i]).index = i
    Else
        LIST(H(A[i]).counter++)  //Increment the counter
EndFor


Bool FOUND = FALSE
For (i = 1 to LIST.length)
    If(LIST[i].counter > n/3)
        Print A[LIST[i].index]
        FOUND = TRUE
EndFor
If (!FOUND)
  Print "There is no candidate with 1/3 of the votes"
```

Brandon Vo

9. (3 bonus points) Think of a good exam/homework question for the material covered in Lecture 7.

What would our run-time be if we used deterministic selection to partition n elements into groups of size k where k < n? How would that run-time be affected if we used quick-sort or merge-sort to arrange the median representatives?