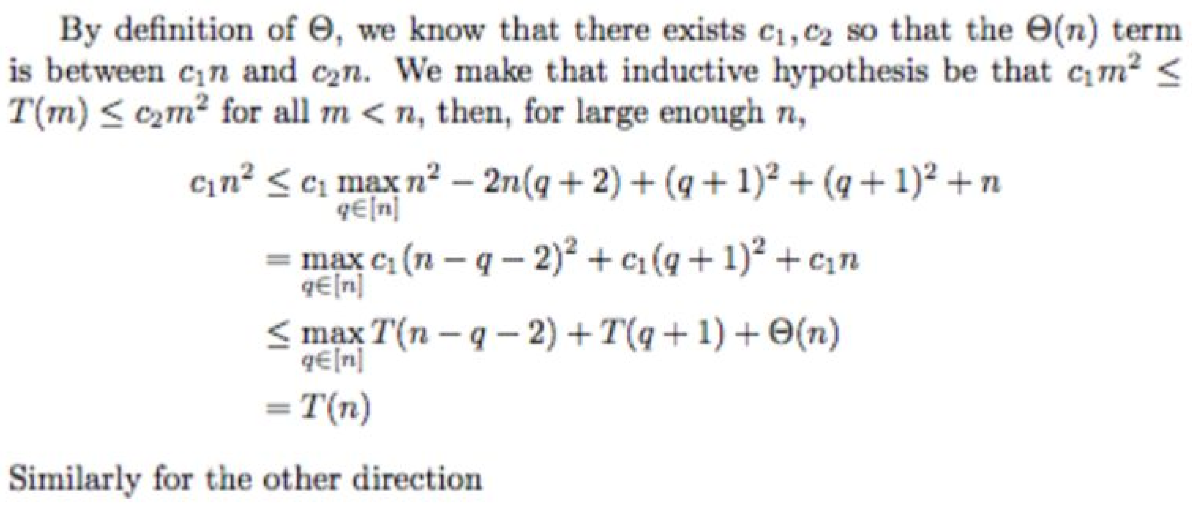
Solution - 6

1.

a. $T(n) = \Theta(n^{\log_2 3})$ –> $a=2$, $b=2$, $f(n) = 1$
b. $T(n) = \Theta(n^{\log_4 5})$ –> $a=5$, $b=4$, $f(n) = n$
c. $T(n) = \Theta(n \log n)$ –> $a=7$, $b=7$, $f(n) = n$
d. $T(n) = \Theta(n^2 \log n)$ –> $a=9$, $b=3$, $f(n) = n^2$
e. $T(n) = \Theta(n^3 \log n)$ –> $a=8$, $b=2$, $f(n) = n^3$
f. $T(n) = \Theta(n^{\log_2 7})$ –> $a=7$, $b=2$, $f(n) = \Theta(n^2)$
g. $T(n) = \Theta(\log n)$ –> $a=1$, $b=2$, $f(n) = \Theta(1)$
h. $T(n) = \Theta(n^2)$ –> $a=5$, $b=4$, $f(n) = \Theta(n^2)$

2.



By definition of Θ, we know that there exists c_1, c_2 so that the $\Theta(n)$ term is between $c_1 n$ and $c_2 n$. We make that inductive hypothesis be that $c_1 m^2 \leq T(m) \leq c_2 m^2$ for all $m < n$, then, for large enough n,

$$\begin{aligned} c_1 n^2 &\leq c_1 \max_{q \in [n]} n^2 - 2n(q+2) + (q+1)^2 + (q+1)^2 + n \\ &= \max_{q \in [n]} c_1 (n-q-2)^2 + c_1 (q+1)^2 + c_1 n \\ &\leq \max_{q \in [n]} T(n-q-2) + T(q+1) + \Theta(n) \\ &= T(n) \end{aligned}$$

Similarly for the other direction

3.

1st A = {6, 19, 9, 5, 12, 8, 7, 4, 11, 2, 13, 21} i = 1 j = 11

2nd A = {6, 2, 9, 5, 12, 8, 7, 4, 11, 19, 13, 21} i = 2 j = 10

3rd A = {6, 2, 9, 5, 12, 8, 7, 4, 11, 19, 13, 21} i = 10 j = 9

4.while the asymptotic complexity is the same, given that there are many duplicates, hoare's partition scheme performs far fewer swaps than Lomuto (what was discussed in class) and is thus more efficient.

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

5.

```
find_max_min(nums):
    if len(nums) == 1:
        return nums[0], nums[0]
    if len(nums) == 2:
        if nums[0] > nums[1]: return nums[0], nums[1]
        else: return nums[1], nums[0]
    max1, min1 = find_max_min(first half of nums)
    max2, min2 = find_max_min(second half of nums)
    return max(max1, max2), min(min1, min2)
```

$T(n) = 2 + 2 * T(n / 2)$ and $T(1) = 0$ and $T(2) = 1$;

So $T(n) = 3n/2-2 < 3n/2$

6. The problem is, if the points are all in a vertical line, imagine there are and only are 4 points in the same vertical line, after every partition, the left set will have 4 points while the right set will

have 0 points. And this recursive partition will keep calling itself again and again while doing nothing. And no output will be given.

7.

If we know the majority element in the left and right halves of an array, we can determine which is the global majority element in linear time.

Here, we apply a classical divide & conquer approach that recurses on the left and right halves of an array until an answer can be trivially achieved for a length-1 array. Note that because actually passing copies of subarrays costs time and space, we instead pass lo and hi indices that describe the relevant slice of the overall array. In this case, the majority element for a length-1 slice is trivially its only element, so the recursion stops there. If the current slice is longer than length-1, we must combine the answers for the slice's left and right halves. If they agree on the majority element, then the majority element for the overall slice is obviously the same1. If they disagree, only one of them can be "right", so we need to count the occurrences of the left and right majority elements to determine which subslice's answer is globally correct. The overall answer for the array is thus the majority element between indices 0 and n.

8.

First, we need to preprocess the input by computing the price changes.

```
PREPROCESS(prices)
changes = new array[prices.length - 1]
for i = 0 to prices.length - 1 {
        changes[i] = prices[i + 1] - A[i]
return changes
```

After we have the price changes array, the original problem becomes finding the maximum subarray sum problem. We can apply the divide and conquer technique to find the best buying and selling time.

We first divide the array into two halves. There are three possibile ways the best transaction can happen:

(1) It can happen on the left half (buy and sell both on the left half)

(2) It can happen on the right half (buy and sell both on the right half)

(3) Or it can happen accross the two halves (buy on the left half and sell on the right half)

After we divide the array, we recursively solve for the left half and the right half. In addition, we also need to solve for the case where transaction happens across the two halves. Finally, we compare the three results, and return the best one.

When the size of the problem becomes small enough (1 element), we can solve it using brute force in O(1) time.

The recurrence relationship is

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

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

The overall time complexity is O(nlogn).

```
MAX_SUBARRAY(A, low, high)
if low == high:    // base case, 1 element
    return (low, high, A[low])
else:
    mid = floor((low + high) / 2)
    (left_low, left_high, left_sum) = MAX_SUBARRAY(A, low, mid)
    (right_low, right_high, right_sum) = MAX_SUBARRAY(A, mid + 1, high)
    (cross_low, cross_high, cross_sum) = MAX_CROSS_SUBARRAY(A, low, mid,
high)
    if left_sum >= right_sum and left_sum >= cross_sum:
        return (left_low, left_high, left_sum)
    else if right_sum >= left_sum and right_sum >= cross_sum:
        return (right_low, right_high, right_sum)
    else:
        return (cross_low, cross_high, cross_sum)
MAX_CROSS_SUBARRAY(A, low, mid, high)
 left_sum = -infinity
sum = 0
for i = mid down to low:
    sum = sum + A[i];
    if (sum > left_sum):
        left_sum = sum
        max_left = i
right_sum = -infinity
sum = 0
for j = mid + 1 to high:
    sum = sum + A[j]
    if (sum > right_sum):
        right_sum = sum
        max_right = j
return (max_left, max_right, left_sum + right_sum)
```

9.

In class, we sort the y-coordinates in each recursive call, in which the solution is T(n) = O(n log2 n). But the y-coordinates could also be sorted before we run the divide and conquer algorithm. We could have two arrays X and Y storing the points as in x-coordinates and y-coordinates monotonically increasing. In the algorithm, the line l that bisects the points into two sets PL and PR with equal number of points. Then we divide the array X into arrays XL and XR, which contain the points of PL and PR respectively, sorted by monotonically increasing x-coordinate. Similarly, divide the array y into arrays YL and YR, which contain the points of PL and PR respectively, sorted by monotonically increasing y-coordinate. By doing this, we could save the time in each recursive call when we have to sort the y-coordinate every single time. The recurrence formula should be T(n) = 2T(n/2) +O(n) and solution could be O(n log n).