Solutions for HW3

1 Problem 1 from Keller and Trotter

1.1 Preamble

The wording for this question is admittedly rather vague. For each part, the question is asking you to do two things:

1. Specify an algorithm to do the given task.
2. Analyze its runtime.

If you used a different algorithm, your answer might be different from the one provided here. However, as long as your algorithm performs the specified task and you've analyzed its runtime correctly, you should receive full marks.

What makes this problem tricky is that we have a list of n integers of size $\leq 100n$. In other words, the size of our numbers and the length of our list both grow with n and both of these will affect the runtime. For instance, mergesort (https://en.wikipedia.org/wiki/Merge_sort) requires at most $O(n \log n)$ comparisons to sort a list of length $O(n)$. However, if our integers are really big, each individual comparison can be quite costly. For instance, if we have two integers of size $O(n)$, then comparing them to see which one is bigger requires $O(\log n)$ time. Why? Well, an integer of size $O(n)$ can be represented using $O(\log n)$ bits. What is the computer doing when it compares two integers? It compares them bit by bit! Thus, the runtime of applying mergesort to a list of $O(n)$ integers each of which has size $O(n)$ is $O(n \log(n)) \cdot O(\log(n)) = O(n(\log n)^2)$.

In the solutions to this problem, we will make the following assumptions about the time required by a computer to perform the following basic tasks. For further information, check out the wikipedia page on the computational complexity of mathematical operations: https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations.

- Checking if two numbers of size $O(n)$ are equal to one another requires $O(\log n)$ time. (Two numbers are equal if they have the same bits at the same positions. A number of size $O(n)$ can be represented using $O(\log n)$ bits, so there are $O(\log n)$ bits to check.)
- Comparing two numbers of size $O(n)$ to see which one is greater requires $O(\log n)$ time. (Again, how do you compare two numbers? You compare them bit by bit and there are at most $O(\log n)$ bits to check.)
- Adding two numbers of size $O(n)$ requires $O(\log n)$ time. (Add the numbers bit by bit, the same way you would add numbers digit by digit in grade school.)
- Subtracting two numbers of size $O(n)$ requires $O(\log n)$ time. (Again, perform the subtraction bit by bit, the same way you would have subtracted two numbers digit by digit in grade school.)
- Multiplying two numbers of size $O(n)$ requires $O((\log n)^2)$ bitwise operations, using the basic algorithm you probably learned in grade school. While all of the algorithms mentioned above were optimal, surprisingly, there are better ways to multiply integers! The first improvement was due to Karatsuba: https://en.wikipedia.org/wiki/Karatsuba_algorithm. The current record is held by an algorithm of Harvey and Van der Hoeven (from 2019!) which can multiply n- bit integers in $O(n \log n)$ time: https://annals.math.princeton.edu/2021/193-2/p04.

- If we have two numbers of size $O(n)$, finding the quotient and remainder when we divide one number by the other, using long division, requires $O((\log n)^2)$ time.

1.2 Solutions

1. (a) For each element of the list, just check whether it is equal to $2n+7$. Checking whether a number of size $O(n)$ is equal to $2n+7$ requires $O(\log n)$ time and our list has $n = O(n)$ elements for a total runtime of $O(n) \cdot O(\log n) = \boxed{O(n \log n)}$.

   Remark: notice that sorting our list won't help us here, since sorting our list via mergesort already requires $O(n(\log n)^2)$ time.

   (b) For each pair of elements (x, y) where x and y are elements of our list, compute $x + y$ and check whether this is equal to $2n + 7$. There are $n^2 = O(n^2)$ such pairs. Computing the sum of x and y for each pair (x, y) requires $O(\log n)$ time, since $x, y \leq 100n$. Similarly, since $x + y \leq 200n$, checking that $x + y$ is equal to $2n + 7$ requires $O(\log n)$ time for a total runtime of:

   $$O(n^2) \cdot (O(\log n) + O(\log n)) = \boxed{O(n^2 \log n)}$$

   Remark: the algorithm described above is the brute force approach to this problem. It is not optimal. In particular, if you sort your list of integers and then apply a little cleverness, you can produce an algorithm that does the prescribed task in $O(n(\log n)^2)$ time.

   (c) The brute force approach to this problem is to compute the product xy for every pair (x, y) of numbers x and y from our list and check if it's equal to $2n+7$. There are $n^2 = O(n^2)$ pairs (x, y) and for each pair (x, y), since $x, y \leq 100n$, we can compute the product xy in $O((\log n)^2)$ time. Checking whether the product xy is equal to $2n + 7$ requires $O(\log n)$ time for a total runtime of $O(n^2) \cdot (O((\log n)^2) + O(\log n)) = \boxed{O(n^2(\log n)^2)}$. That said, the hint for this problem suggested that we first sort our list, implying that the book wants us to do something more clever to improve the runtime.

   To improve on the runtime above, we first sort our list using mergesort to get the sorted list $(a_1, a_2, a_3, ..., a_n)$. As discussed in the preamble, this takes at most $O(n(\log n)^2)$ time. From there, starting with a_1, then going on to a_2, then a_3, etc. for each element a_i, we perform long division to find the quotient q_i and remainder r_i that we get when we divide $2n + 7$ by a_i. That is, we find q_i and r_i such that $q_i \cdot a_i + r_i = 2n + 7$ where q_i and r_i are nonnegative integers such that $r_i < a_i$. Since the numbers involved are of size $O(n)$, this takes at most $O((\log n)^2)$ time. If $r_i \neq 0$, then a_i does not divide $2n + 7$ and we move on to the next integer in our list. If $r_i = 0$, then a_i divides $2n + 7$ and it is possible that the quotient q_i is somewhere in our list so that $2n + 7$ is the product of two elements in our list, namely a_i and q_i. We use binary search (https://en.wikipedia.org/wiki/Binary_search_algorithm) to check if q_i is on our list. For a sorted list of integers of length $O(n)$, binary search requires $O(\log n)$ comparisons to find out if a particular integer is in our list. Since our integers are of size $O(\log n)$, each comparison requires $O(\log n)$ time. Thus, checking whether q_i is on our list requires $O((\log n)^2)$ time. If at some point we find that $2n + 7$ is the product of two integers on our list, we stop. Otherwise, we keep going until we've gone through all the elements a_i of our list. Putting all this together, the runtime of this algorithm is:

   $$O(n(\log n)^2) + O(n) \cdot (O((\log n)^2) + O((\log n)^2)) = \boxed{O(n(\log n)^2)}$$

   Remark: the reason for the improved runtime of this second algorithm are twofold. Firstly, going through the n integers of our list and computing n/a_i for each i is way more efficient than computing $x \cdot y$ for each distinct pair of integers on our list, of which there are n^2. Secondly, sorting

our list allowed us to use binary search to figure out if a given integer is in our list, rather than just going through the list element by element.

(d) Observe that the numbers in our list lying between i and $2n+7+i$ for some i is equivalent to the difference between the maximal and minimal integers in our list being at most $2n+7$. Thus, it suffices to find the maximal and minimal integer in our list and compute their difference. Finding the maximal integer in our list requires $n-1$ comparisons. Since the numbers involved are of size $O(n)$, each comparison requires $O(\log n)$ time. Similarly, finding the minimal integer in our list requires $n-1$ comparisons where each comparison requires $O(\log n)$ time. Finally, computing the difference between the maximal and minimal integers in our list requires $O(\log n)$ time for a total runtime of:

$$O(n)\cdot O(\log n) + O(n)\cdot O(\log n) + O(\log n) = \boxed{O(n\log n)}$$

(e)

(f) Finding the number of primes on our list is a 3 - step process:

i. First, we build a list of all the primes $\leq 100n$. We will elaborate on how to do this shortly. By definition, each element on the resulting list has size at most $100n$. By the prime number theorem (https://en.wikipedia.org//wiki/Prime_number_theorem), this list has length $O(\frac{n}{\log n})$.

ii. Next, we use mergesort to sort our list of n integers, each of which has size $O(n)$. As we discussed earlier, this requires $O(n(\log n)^2)$ time.

iii. Now that we've sorted our list from part 2, we can search it using binary search. For each prime in the list from part 1, we check if it's on the list from part 2 using binary search. This costs $O((\log n)^2)$ for each element in our list of prime numbers, of which there are $O(\frac{n}{\log n})$ for a total runtime of $O(n\log n)$.

Now let's talk about the runtime of step 1. To produce a list of all the prime numbers $\leq 100n$ we use the classical sieve of Eratosthenes (https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes). That is, let P_n be the list of prime numbers $\leq n$, arranged in increasing order. P_2 is just (2). For $n \geq 3$ we do the following: if n is not divisible by any of the elements of P_{n-1}, we append n to P_{n-1}. The resulting list is P_n. If n IS divisible by one of the elements of P_{n-1}, then n is not prime. In this case, $P_n = P_{n-1}$. What's the runtime of this algorithm? Well, at the kth stage of this process we take k and for each element q of P_{k-1} we compute the remainder of dividing k by q using long division. Since P_{k-1} is a list of length $O(\frac{k}{\log k})$, each of whose elements has size $O(k)$, this requires at most $O(k\log k)$ time. Summing over n, the time it takes this algorithm to build a list of all the primes $\leq n$ is:

$$\sum_{k=2}^{n} O(k\log k) = O\Big(\sum_{k=2}^{n} k\log k\Big) = O\Big(\int_2^n x\log x dx\Big) = O(n^2\log n)$$

Putting all of this together we get a total runtime for our algorithm of:

$$O(n^2\log n) + O(n(\log n)^2) + O(n\log n) = \boxed{O(n^2\log n)}$$

Remark: our analysis of the runtime of the Sieve of Eratosthenes was admittedly pretty crude. In particular, if a number of size $O(k)$ is composite, its smallest prime divisor is usually going to be much smaller than k, so most of the time, we'll only need to compute the remainder of k modulo q for a small number of integers q in our list of primes. As per the wikipedia article linked to above, more careful analysis of this algorithm shows that the runtime is actually on the order of $O(n\log n\log\log n)$. Thus, sorting our list is actually the most time- consuming step, not building this list of primes.

(g) First sort our list of n integers so that the entries are in increasing order. For each pair of integers (x, y) where x and y are entries in our list, compute $x + y$ and using binary search, check if $x + y$ is in our list. We do this until we find a pair (x, y) so that $x + y = z$ for some integer z in our list, in which case we stop. Otherwise, we keep going until we exhaust all possible pairs (x, y). As far as analyzing the runtime of this algorithm,

- Sorting our list using mergesort requires $O(n(\log n)^2)$ time.
- There are $n^2 = O(n^2)$ pairs of integers (x, y) where x, y are entries in our list.
- For each pair (x, y), $O(\log n)$ time is required to compute $x + y$, since the entries of our list are of size $O(n)$. From there, finding out whether $x + y$ is in our list using binary search requires $O(\log n)$ comparisons. Since the numbers involved are of size $O(n)$, each comparison requires $O(\log n)$ time.

In summa, the runtime of this algorithm is:

$$O(n(\log n)^2) + O(n^2) \cdot (O(\log n) + O((\log n)^2)) = \boxed{O(n^2(\log n)^2)}$$

(h) First sort our list of n positive integers so that the entries are in increasing order. Let $(a_1, a_2, ..., a_n)$ be the resulting sorted list. For each index i, we multiply a_i by itself to get the new list $(a_1^2, a_2^2, ..., a_n^2)$. This can be done in $O(n(\log n)^2)$ time, since multiplying two numbers of size $O(n)$ can be done in $O((\log n)^2)$ time, and we are just doing this n times. Observe now that finding integers x, y, z in our original list $(a_1, a_2, ..., a_n)$ such that $x^2 + y^2 = z^2$ is equivalent to the problem of finding integers x, y, z in our new list $(a_1^2, a_2^2, ..., a_n^2)$ so that $x + y = z$. But this is exactly the problem we addressed in part (g)! The only difference is now the entries of our list are of size $O(n^2)$ instead of $O(n)$, but this difference turns out to be unimportant since the contribution of the size of our integers to the runtime depends on the number of bits that is required to represent them. Whether our numbers are of size $O(n)$ or $O(n^2)$, both can be represented by $O(\log n)$ bits. Thus, applying our algorithm from part (g) to our new list allows us to solve this problem in time:

$$O(n(\log n)^2) + O(n^2(\log n)^2) = \boxed{O(n^2(\log n)^2)}$$

(i) First sort our list of n integers so that the entries are in increasing order. For each pair of integers (x, y) where x and y are entries in our list, compute xy and using binary search, check if xy is in our list. We do this until we find a pair (x, y) so that $xy = z$ for some integer z in our list, in which case we stop. Otherwise, we keep going until we exhaust all possible pairs (x, y). As far as analyzing the runtime of this algorithm,

- Sorting our list using mergesort requires $O(n(\log n)^2)$ time.
- There are $n^2 = O(n^2)$ pairs of integers (x, y) where x, y are entries in our list.
- For each pair (x, y), we can compute xy in $O((\log n)^2)$ time, since the entries of our list are of size $O(n)$. From there, finding out whether xy is in our list using binary search requires $O(\log n)$ comparisons. Since the numbers involved all have $O(\log n)$ bits, each comparison requires $O(\log n)$ time.

In summa, the runtime of this algorithm is:

$$O(n(\log n)^2) + O(n^2) \cdot (O((\log n)^2) + O((\log n)^2)) = \boxed{O(n^2(\log n)^2)}$$

(j) To answer this question, we first need to understand the computational complexity of computing x^y for a pair of integers x and y. Write out y in binary:

$$y = \sum_{i=0}^{m} a_i 2^i$$

where $a_i \in \{0, 1\}$ for all i and $m = \lfloor \log_2 x \rfloor$. For each $i \in \{0, 1, 2, ..., m\}$ we compute x^{2^i} recursively by multiplying $x^{2^{i-1}}$ and $x^{2^{i-1}}$. As discussed earlier, we can do this in $O((\log x^{2^{k-1}})^2) =$

$O(4^k(\log x)^2)$ time. Thus, the total cost of computing $x^{2^1}, x^{2^2}, x^{2^3}, ..., x^{2^m}$ in this way is:

$$\sum_{i=1}^{m} O(4^i(\log x)^2) = O\Big(\sum_{i=1}^{m} 4^i(\log x)^2\Big) = O\Big(\Big(\frac{4^{m+1}-1}{3}\Big)(\log x)^2\Big) = O(4^m(\log x)^2)$$
$$= O(4^{\lfloor \log_2 y \rfloor}(\log x)^2) = O(y^2(\log x)^2)$$

From here, we compute x^y by multiplying powers of x as follows:

$$x^y = x^{\sum_{i=0}^{m} a_i 2^i} = x^{2^{a_m}}(...x^{2^3 a_3}(x^{2^2 a_2}(x^{2^1 a_1} x^{a_0}))...)$$

The brackets in the product above indicate the order in which we multiply our powers of x. In particular, notice that at the k^{th} stage, we are multiplying two numbers of size $O(x^{2^k})$ which, as discussed earlier, can be done in $O((\log x^{2^k})^2) = O(4^k(\log x)^2)$ time. Thus, the total cost of computing this product is:

$$\sum_{k=1}^{m} O(4^k(\log x)^2) = O(y^2(\log x)^2)$$

In summa, given two positive integers x and y, we can compute x^y in $O(y^2(\log x)^2)$ time. With this in mind, we can now deal with the problem at hand.

- First, we sort our list of n positive integers, each of which has size $\leq 100n$ using mergesort.This requires $O(n(\log n)^2)$ time. Let $(b_1, b_2, b_3, ..., b_n)$ denote the resulting sorted list.
- For each integer x on our list, compute $x^{b_1}, x^{b_2}, x^{b_3}...$ until we arrive at an integer b_k so that $x^{b_k} > 100n$, meaning that x^{b_i} cannot be on our list for any $i \geq k$. Since $b_1 \leq b_2 \leq ... \leq b_{k-1}$ where $x^{b_{k-1}} \leq 100n \implies b_{k-1} = O(\frac{\log n}{\log x})$, the cost of computing x^{b_i} for any $i < k$ is:

$$O(b_i^2(\log x)^2) = O\Big(\Big(\frac{\log n}{\log x}\Big)^2(\log x)^2\Big) = O((\log n)^2)$$

- Having computed x^{b_i} for some $i < k$, we check whether or not this number is on our list using binary search. If we find some index r so that $x^{b_i} = b_r$ we stop, having answered our question in the affirmative. Otherwise, we keep going. Since our list has length n and all the numbers involved are of size $\leq 100n$, the runtime of this is $O((\log n)^2)$.
- For each fixed x, we need to compute x^{b_i} for at most n indices i and for each i, check if x^{b_i} is on our list. Thus, every element x of our list contributes at most $n \cdot \big(O((\log n)^2) + O((\log n)^2)\big) = O(n(\log n)^2)$ to the runtime. Since there are n elements on our list, this gives us a total runtime of $\boxed{O(n^2(\log n)^2)}$.

(k) Observe that if x and y are positive integers, x^y is prime if and only if x is prime and $y = 1$. Thus, to determine if there is a pair (x, y) on our list so that x^y is prime, we first check whether 1 is on our list. This requires $O(n)$ time, since there are n numbers to check and we can find out if a number is equal to 1 in $O(1)$ bitwise comparisons. From there, we check whether there is a prime on our list. Using the algorithm from part (f), we can do this in $O(n^2 \log n)$ time. In summa, the runtime of this algorithm is:

$$O(n) + O(n^2(\log n)) = \boxed{O(n^2 \log n)}$$

(l)

(m)

(n)

(o)

2 Problems 2-3 from Keller and Trotter

2. If you put $mn+1$ pigeons into n holes, one of these holes must contain at least $(m+1)$ pigeons. (If this were not the case, then each hole would contain at most m pigeons so the total number of pigeons would be at most $mn < (mn+1)$.) □

3. i If $X = \{1,2,3,4,5\}$ it IS possible to place the 2- element subsets of X into two holes so that for any 3- element subset $\{a,b,c,\} \subseteq X$, $\{a,b\},\{b,c\}$ and $\{a,c\}$ aren't all in the same hole. In fact, there are multiple ways to do this. One way to do this is as follows: place $\{1,2\},\{1,3\},\{2,4\},\{2,5\},\{3,5\}$ in the first hole and $\{1,4\},\{1,5\},\{2,3\},\{3,4\},\{4,5\}$ in the second hole. Then for any 3- element subset $\{a,b,c\} \subseteq \{1,2,3,4,5\}$ you can check that $\{a,b\},\{b,c\}$ and $\{a,c\}$ are not all in the same hole.

   ii If $X = \{1,2,3,4,5,6\}$, this is not possible. Why? Well, there are five 2- element subsets of X containing 1: $\{1,2\},\{1,3\},\{1,4\},\{1,5\}$ and $\{1,6\}$. By the pigeonhole principle, one of our holes must contain at least three of these subsets. Without loss of generality, suppose that $\{1,2\},\{1,3\}$ and $\{1,4\}$ are all placed in the first hole. If any of subsets $\{2,3\},\{2,4\}$ or $\{3,4\}$ is also in the first hole, we have that there is a 3- element subset $\{a,b,c\} \subseteq X$ so that $\{a,b\},\{b,c\}$ and $\{a,c\}$ are all in the same hole. Thus, to avoid this, $\{2,3\},\{2,4\}$ and $\{3,4\}$ must all be placed in the other hole. However, in this case, we see that the other hole contains all of the 2- element subsets of $\{2,3,4\}$. Thus, we see that however we place the 2- element subsets of X into two holes, one hole will contain all of the 2- element subsets of some 3- element set $\{a,b,c\} \subseteq X$. □

   Remark: now that you know some graph theory, notice that problem 3 can actually be reinterpreted as a graph coloring problem. Namely, given the finite set $X = \{1,2,3,...,n\}$, we can think of the 2- element subsets of X as edges of the complete graph on n vertices. Placing these 2- element subsets into two holes can be interpreted as coloring the edges of K_n one of two colors. Namely, given an edge $\{a,b\}$, we color it red if $\{a,b\}$ is placed in the first hole. Otherwise, we color it blue. In this language, question 2 is asking you the following: is it possible to color the edges of K_5 (in part (i)) and K_6 (in part (ii)) so that there are no monochromatic triangles?

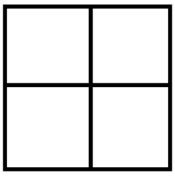
3 Problems 1-5 from the Pigeonhole Principle Handout

1. A quick Google search reveals that the current population of Toronto is around 6.372 million. Meanwhile, the number of hair follicles on the head of a healthy adult human ranges between $90,000$ and $150,000$. Thus, by the pigeonhole principle, there are at least:

$$\left\lceil \frac{6.372 \times 10^6}{1.5 \times 10^5} \right\rceil = 43$$

   people in Toronto with the same number of hairs on their head.

2. Subdivide the unit square into four squares with side length $1/2$ as so:



   If a pair of points lie in the same subsquare of the unit square, the distance between them is at most $\sqrt{2}/2$. If we choose 5 points in the unit square, by the pigeonhole principle, two of them must lie in the same subsquare with side length $1/2$. The distance between these two points is then at most $\sqrt{2}/2$. □

3. Assume that acquaintance is mutual. That is, if person A knows person B, then person B knows person A. In this case, we say that A and B are acquaintances. We claim that among the 20 people present at the party, there are two that have the same number of acquaintances among those present. To prove this, we consider two cases:

   Case 1: There is at least one person who knows everyone else.

   Since acquaintance is mutual, if there is at least one person who knows everyone else at the party, it follows that everyone at the party knows at least one person. Thus, each person at the party has between 1 and 19 acquaintances among the other 19 people at the party. Since there are 20 people at the party, by the pigeonhole principle, there must be $\lceil \frac{20}{19} \rceil = 2$ people who have the same number of acquaintances present at the party.

   Case 2: There is no person at the party who knows every other person at the party.

   In this case, every person at the party has between 0 and 18 acquaintances among the other 19 people at the party. By the pigeonhole principle, there must be $\lceil \frac{20}{19} \rceil = 2$ people with the same number of acquaintances. □

4. Consider the list of numbers:

$$a_1, \quad a_1 + a_2, \quad a_1 + a_2 + a_3, \quad a_1 + a_2 + a_3 + a_4, \quad a_1 + a_2 + a_3 + a_4 + a_5$$

   These are the sums of the elements in the subsets $\{a_1\}, \{a_1, a_2\}, \{a_1, a_2, a_3\}, \{a_1, a_2, a_3, a_4\}$ and $\{a_1, a_2, a_3, a_4, a_5\}$. If one of these numbers is divisible by 5, then we've found a subset of $\{a_1, a_2, a_3, a_4, a_5\}$ whose sum is divisible by 5. Otherwise, each of these numbers has remainder $1, 2, 3$ or 4 modulo 5. Since we have 5 numbers and 4 possible values for the remainder modulo 5, by the pigeonhole principle, two of these numbers must have the same remainder modulo 5 and so their difference is divisible by 5. But the difference between any pair of numbers in the list above is itself the sum of the elements of some subset of $\{a_1, a_2, a_3, a_4, a_5\}$. Thus, we've found a subset of $\{a_1, a_2, a_3, a_4, a_5\}$ whose sum is divisible by 5. □

5. Given $n+1$ positive integers of size at most $2n$, we want to show that we have two integers so that one divides the other. Notice that we can assume our $n+1$ integers are distinct: were this not the case, since any number divides itself, we immediately have a pair of positive integers in our set so that one divides the other. Split the set $\{1, 2, 3, ..., 2n\}$ into subsets $S_{k,n}$ defined as follows:

$$S_{k,n} := \{m \in \mathbb{N} : m = (2k-1)2^r \text{ for some } r \in \mathbb{N}_0 \text{ and } m \leq 2n\}$$

   where $k \in \{1, 2, ..., n\}$. That is, each $S_{k,n}$ is the set of numbers between 1 and $2n$ that can be written as $(2k-1)$ multiplied by a power of 2 and we can write $\{1, 2, 3, ..., 2n\}$ as a disjoint union of sets $S_{k,n}$:

$$\{1, 2, 3, ..., 2n\} = \bigsqcup_{k=1}^{n} S_{k,n}$$

   Suppose $A \subseteq \{1, 2, 3, ..., 2n\}$ is a set with $n+1$ elements. If we take the elements of A as our pigeons and the $S_{k,n}$'s as our holes, the pigeonhole principle tells us that for some positive integer $k \leq n$, $S_{k,n}$ contains at least $\lceil \frac{n+1}{n} \rceil = 2$ elements of A. However, by the definition of $S_{k,n}$, if $a, b \in S_{k,n}$ where $a \leq b$ then a divides b. Thus, we see that any set of $n+1$ positive integers of size at most $2n$ has two elements so that one divides the other.

   Remark: while this problem is on a pigeonhole principle handout, which suggests that this is the technique you should try to use, it is probably easier to solve it by induction. In particular, if you couldn't figure out how to solve this problem on your own or had to look up the answer on Stackexchange, try see if you can solve it using induction.