- See Practice Problems Sets 1 for instructions.
- Treat all arrays as 0-indexed. You may assume the following in your analyses:
  - Storing an integer takes constant space, irrespective of how large the integer is. This is clearly not true (even in theory), but our focus in these problems is not on the intricacies of dealing with large integers. So we make this assumption for the sake of simplifying the analysis.
  - Each operation—arithmetic, comparison, etc.—between a pair of integers takes constant time, irrespective of how large the integers are.
  - Accessing a single array element takes constant time, irrespective of the length of the array.
- Clearly state any other assumption that you need.
- When arguing the correctness of loops, you *need not* use loop invariants (But of course, you *may*!), unless *specifically* asked to do so. You must always clearly explain why each loop correctly does what you want it to do.

1. Refer to the sorting algorithm in your solution to Problem 8 in Problem Set 2.
   (a) Obtain a good asymptotic upper bound (that is, of the form $T(n) = \mathcal{O}(f(n)$, for some function f) on the *worst-case* running time of this algorithm on an input list of size n.
   (b) Obtain a good asymptotic upper bound on the *best-case* running time of this algorithm on an input list of size n.

2. Read up on what a palindrome is, if required. Your solution to parts (c) and (d) must consist of deriving recurrence relations and solving them using the "estimate and verify" method that we saw in class.
   (a) Write the pseudocode for a **recursive** function IsPalindrome(*string*) that checks if the given string is a palindrome. (*Hint:* You already did something like this for Problem Set 1.)
   (b) Prove using induction that your pseudocode of part (a) is correct. (*Hint:* Ditto.)
   (c) Obtain a good asymptotic upper bound on the worst-case running time of your algorithm of part (a).
   (d) Obtain a good asymptotic upper bound on the best-case running time of your algorithm of part (a).

3. Refer to your algorithm FindLargest from Problem Set 1. For each part below, your analysis must consist of deriving recurrence relations and solving them using the "estimate and verify" method that we saw in class.
   (a) Obtain a good asymptotic upper bound on the worst-case running time of your algorithm, in terms of the number of numbers in the input.

(b) Obtain a good asymptotic upper bound on the best-case running time of your algorithm, in terms of the number of numbers in the input.

4. Read up the definition of the Longest Common Substring problem.

   (a) Write the pseudocode for an algorithm that solves this problem for two input strings x, y, starting with a *recursive* solution for the following simpler version: Given (x, y, i, j, r) as input where i, j, r are integers, check whether the r-length substrings of x, y that start at indices i, j, respectively, are identical. That is, whether $x[i]x[i+1]\cdots x[i+r-1]$ is identical to $y[j]y[j+1]\cdots y[j+r-1]$.

   (b) Let m, n be the lengths of inputs x, y, respectively. Analyze your algorithm of part (a) to obtain a good asymptotic upper bound on the time it takes to solve Longest Common Substring, in terms of m and n.

5. Prove the following statement using the definition of the $\mathcal{O}()$ notation that we saw in class:

   Claim

   Let $f(n), g(n)$, and $h(n)$ be functions from non-negative integers to real numbers. If $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(h(n))$, then $f(n) = \mathcal{O}(h(n))$.

6. Prove or disprove:

   (a) $3n = \mathcal{O}(2n)$

   (b) $\log_2 3n = \mathcal{O}(\log_2 2n)$

   (c) $2^{3n} = \mathcal{O}(2^{2n})$

   (d) $n^{\log_2 \log_2 n} = \mathcal{O}(2^{\sqrt{\log_2 n}})$

   (e) $2^{\sqrt{\log_2 n}} = \mathcal{O}(n^{\log_2 \log_2 n})$

7. For each of the following recurrences, unroll the recurrence to come up with an estimate $f(n)$ that satisfies $T(n) = \mathcal{O}(f(n))$. In each case, verify your estimate by induction.

   You may assume bounds of the form $T(n') \leq c'$ where n', c' are fixed constants of your choice. That is, you may assume constant upper bounds for inputs of up to some constant size.

   (a) $T(n) = T(n/2) + c$, where c is a fixed positive integer.

   (b) $T(n) = T(n/2) + 3\log_2 n$

   (c) $T(n) = 2T(n/2) + 5n\sqrt{n}$

   (d) $T(n) = 2T(n/2) + \frac{n}{\log_2 n}$

8. Consider the following problem, which occurs as a sub-problem in Merge Sort:

Merge

- Input: Integers $\ell \geq 1, r \geq 1$ and arrays L, R containing ℓ, r integers respectively. Each of L, R is sorted in non-decreasing order.
- Output: An array M that contains all the elements present in L and R (with repetitions, if any), and is sorted in non-decreasing order. This is the array that we get when we "merge" L and R.

Write the pseudocode for a function $Merge(\ell, r, L, R)$ that solves the above problem in time $\mathcal{O}(\ell + r)$, in the following way:

(a) First, come up with suitable notions of (i) a partial solution[1], and (ii) a small step that takes you from this partial solution a small bit towards the complete solution. Convince yourself that these two notions make sense.

(b) Write the pseudocode for a function—say, $MergeHelper(\dots)$ that solves the small step. Note that the set of arguments to this function may be different from the set of arguments to $Merge()$.

(c) "Bootstrap" your solution to part (b) to arrive at the pseudocode for $Merge(\ell, r, L, R)$.

(d) Use one or more loop invariant(s) to prove that your $Merge(\ell, r, L, R)$ function of part (c) is correct.

(e) Argue that your version of $Merge(\ell, r, L, R)$ runs within the required time bound, in the worst case.

The pseudocode that you come up with in this manner does not *have* to be identical to the version that we saw in class; it just has to be correct!

9. (a) Write the pseudocode for a function $MergeSort(A)$ that sorts array A using Merge Sort, and invokes the function $Merge(\ell, r, L, R)$ for the merge step.

   (b) Illustrate how your pseudocode from part (a) works on the input array A given below, by listing the *arguments and return values* of all the calls to the MergeSort and Merge functions that result from the call MergeSort(A), *in the order that these calls are made.*

   The input array is

$$A = [91, 24, 13, 45, 41, 38, 27, 23, 96, 79]$$

10. (a) Convert your pseudocode for $MergeSort(A)$ from part (a) of the previous question to a Python program. Generate a large number of random arrays and test that your code correctly sorts all of them.

    (b) Include print() statements in your Python functions of part (a), to print out the argument and return value of each call to the MergeSort and Merge functions. Give the array

[1] "Someone else has solved *most*, but *not all*, of the problem for me."

$$A = [91, 24, 13, 45, 41, 38, 27, 23, 96, 79]$$

as input to this modified *MergeSort*() function. Does the output exactly match your solution to part (b) of the previous question?

11. Let A be an array with n integer elements, where each integer is between 1 and cn for a fixed constant c.

    (a) Write the pseudocode for a function $HeavySubSet(A, d)$ that takes A and a positive integer d as arguments, runs in $\mathcal{O}(n)$ time, and returns a list of all those integers that appear at least $\lfloor \frac{n}{d} \rfloor + 1$ times in A. Each such integer must be present exactly once in the returned list. If there is no such integer in A, then $HeavySubSet(A, d)$ should return an empty list.

    - *Hint:* As is usually the case, it helps to think of a simpler problem first. For instance: can you think of an $\mathcal{O}(n)$-time algorithm that takes A, d, and an element r of A as arguments, and figures out if r appears at least $\lfloor \frac{n}{d} \rfloor + 1$ times in A? Once you have solved this simpler version, try to think of a way of "lifting" your solution to one for the original problem, without blowing up the running time by more than a constant factor.

    (b) Argue that your algorithm $HeavySubSet(A, d)$ of part (a) is correct.

    (c) Show that your algorithm of part (a) solves the problem in $\mathcal{O}(n)$ time.

12. Let A be an array with n integer elements.

    (a) Write the pseudocode for a function $HeavySubSet(A, d)$ that takes A and a positive integer d as arguments, runs in $\mathcal{O}(n \log_2 n)$ time, and returns a list of all those integers that appear at least $\lfloor \frac{n}{d} \rfloor + 1$ times in A. Each such integer must be present exactly once in the returned list. If there is no such integer in A, then $HeavySubSet(A, d)$ should return an empty list.

    - *Hint:* Solve the same sub-problem as in the hint for question 4(a), but for this type of input. Once you have solved this simpler version, try to think of a way of "lifting" your solution to one for the original problem, without blowing up the running time by more than a *logarithmic* (in n) factor. You may want to try divide-and-conquer (in some form ...) to do this lifting.

    (b) Argue that your algorithm $HeavySubSet(A, d)$ of part (a) is correct.

    (c) Show that your algorithm of part (a) solves the problem in $\mathcal{O}(n \log_2 n)$ time.

13. Let C be an array with n images of cats that took part in a beauty contest for cats. The same cat can have many different images (of different poses/attire, say) present in C. Each image of a given cat appearing in C corresponds to a distinct vote that the cat got in the contest. You have access to a function $SameCat(x, y)$ which takes two such images x, y and tells whether the two images correspond to the same cat, or not. This function takes constant time to do one such comparison.

(a) Write the pseudocode for a divide-and-conquer algorithm *QueenCat*(C) that takes C as the argument, runs in $\mathcal{O}(n \log_2 n)$ time, and returns one image of a cat which (the cat, not necessarily the same image) appears at least $\lfloor \frac{n}{2} \rfloor + 1$ times in C. If there is no such cat in C, then *QueenCat*(C) should return "None".

(b) Argue that your algorithm *QueenCat*(C) of part (a) is correct.

(c) Show that your algorithm of part (a) solves the problem in $\mathcal{O}(n \log_2 n)$ time.

14. From the previous question you know how to find the *majority element*—if one exists—in an input array of size n (and no constraints on its elements, except for constant-time equality checks), in $\mathcal{O}(n \log_2 n)$ time using divide-and-conquer. It turns out that we can in fact solve this problem in *linear* time and *constant extra*[2] *space* **without** using divide-and-conquer. Coming up with such an algorithm requires a non-trivial amount of creativity, and the end result looks like magic.

(a) Read up on the Boyer-Moore majority vote algorithm and the argument for its correctness at, e.g., its Wikipedia page (or elsewhere). Make sure that you understand *why* the algorithm works.

(b) Let C be the array of n cat images from the previous question. Consider the following operation performed on C: as long as there are images x, y, z of three *distinct*[3] cats in L, remove such a set $\{x, y, z\}$ from C. Let $C^\star$ be the array remaining, once this operation can no longer be performed. Prove that if a cat C_∞ had at least $\lfloor \frac{n}{3} \rfloor + 1$ of its images present in the original array C, then it will have at least one image present in the final array[4] $C^\star$.

(c) Write the pseudocode for an algorithm that adapts the Boyer-Moore majority vote algorithm to find all cats that appear at least $\lfloor \frac{n}{3} \rfloor + 1$ times in an input list C of n cat images, in $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ extra space. Your algorithm should output a list with one image for each such distinct cat in C. If there is no such cat in C, then the algorithm should output the empty list. As in the previous question, assume that you have access to the *SameCat*(x, y) function that returns in constant time.

Argue that your algorithm correctly solves the problem, and that it runs in linear time and needs only constant extra space.

15. Let A be an array containing n integers. For $0 \leq i \leq j < n$ we use $A[i : j]$ to denote the *sub-array* $[A[i], A[i+1], \ldots, A[j]]$. Note that this is **different** from the Python convention for lists.

The Maximum Sub-array problem has such an array A as input, and asks for the number

$$\max_{0 \leq i \leq j < n} \sum_{i \leq k \leq j} A[k].$$

[2] That is, over and above the space required for storing the input.

[3] That is, for which *SameCat*(x, y), *SameCat*(y, z), *SameCat*(x, z) all return "No".

[4] In particular, this means that if $C^\star$ is empty then there is no cat whose images appear at least $\lfloor \frac{n}{3} \rfloor + 1$ times in the starting list C.

That is, the goal is to compute the maximum sum, taken over all sub-arrays of A, of all the elements in a sub-array of A.

(a) Write the pseudocode for an algorithm that solves this problem in $\mathcal{O}(n^2)$ time. Argue that your algorithm is correct and that it runs in $\mathcal{O}(n^2)$ time.

(b) We can use divide-and-conquer to improve on the running time of part (a). Towards this, first write the pseudocode for an algorithm that solves the following simpler problem in $\mathcal{O}(n)$ time: Given an index $0 \leq i < n$, find the maximum sum of a sub-array that includes the index i. Argue that your algorithm is correct, and that it runs in $\mathcal{O}(n)$ time.

(c) Write a divide-and-conquer algorithm that solves the Maximum Sub-array problem in $\mathcal{O}(n \log_2 n)$ time. Argue that your algorithm is correct, and that it runs in $\mathcal{O}(n \log_2 n)$ time.

16. It turns out that we can in fact solve the Maximum Sub-Array problem as well in *linear* time and *constant extra space* **without** using divide-and-conquer. As with the majority-finding algorithm, coming up with such an algorithm requires a non-trivial amount of creativity, and the end result again looks like magic. Read up on Kadane's algorithm and its correctness at, e.g., its Wikipedia page (or elsewhere). Make sure that you understand *why* the algorithm works.

17. Let A be an array containing n integers, *sorted* in non-decreasing order. We want to figure out the number of times that a given integer x appears in A. The input to this problem consists of A and x, and the required output is the number of times—this could be zero as well—that x appears in A.

(a) Write the pseudocode for an algorithm Counter(A, x) that solves this problem in $\mathcal{O}(n)$ time. Argue that your algorithm is correct and that it runs in $\mathcal{O}(n)$ time.

(b) Perhaps surprisingly, we can use divide-and-conquer to improve on the running time of part (a).

Towards this, first modify the pseudocode for Binary Search[5] to get an algorithm which, if x is present in A, will find—in $\mathcal{O}(\log_2 n)$ time—the index i such that $A[i]$ is the *last* occurrence of x in A.

Now write the pseudocode for an algorithm SmartCounter(A, x) that solves the problem in $\mathcal{O}(\log_2 n)$ time. Argue that your algorithm is correct, and that it runs in $\mathcal{O}(\log_2 n)$ time.

18. Let $A[0 \ldots (n-1)]$ be an array containing n bits, where the first t bits are 0s, and the remaining bits are all 1s. That is, $A[t]$ is the first location in A that contains a 1, and $A[t-1]$ is the *last* location that contains a 0. We want to figure out the value of the index t. The input to this problem consists of the array A, and the required output is the number t.

(a) Write the pseudocode for an algorithm SimpleFindT(A) that solves this problem in $\mathcal{O}(n)$ time. Argue that your algorithm is correct and that it runs in $\mathcal{O}(n)$ time.

[5] Read up on the Binary Search algorithm—from the above link, and/or from elsewhere—and make sure that you understand how/why it works.

(b) Use divide-and-conquer to design an algorithm SmartFindT(A) that solves this problem in $\mathcal{O}(\log_2 n)$ time. Write the pseudocode for SmartFindT(A). Argue that your algorithm is correct and that it runs in $\mathcal{O}(\log_2 n)$ time.

(c) It turns out that we can use divide-and-conquer in a somewhat unusual—and clever—fashion to *further* improve on the running time of part (b).

Write the pseudocode for a divide-and-conquer algorithm CleverFindT(A) that solves this problem in $\mathcal{O}(\log_2 t)$ time. Note that this running time is *independent* of the input size n, and only depends on the length of the prefix of 0s.

Argue that your algorithm is correct and that it runs in $\mathcal{O}(\log_2 t)$ time.

19. Let $A[0 \ldots (n-1)]$ be an array that contains n integers. We want to find the *smallest* absolute difference between two elements at distinct positions in A. That is, we want to find

$$\min_{0 \le i < j < n} |A[i] - A[j]|.$$

Note that this number:

- Is always a non-negative integer; and,
- Is zero if and only if there are two identical elements at different locations in A.

(a) Write the pseudocode for a procedure AbsDiff(x, y) that takes two integers x, y as arguments and returns their absolute difference $|x - y|$.

(b) Write the pseudocode for an algorithm SimpleSmallestDiff(A) that solves the problem in $\mathcal{O}(n^2)$ time. The algorithm should return this smallest difference. Argue that your algorithm is correct and that it runs in $\mathcal{O}(n^2)$ time.

(c) With some pre-processing of the array A we can find this smallest difference much faster. Write the pseudocode for an algorithm SmartSmallestDiff(A) that solves the problem in $\mathcal{O}(n \log_2 n)$ time. The algorithm should return this smallest difference. Argue that your algorithm is correct and that it runs in $\mathcal{O}(n \log_2 n)$ time.

> The remaining questions are links to programming problems from LeetCode. Solve each of them using an algorithm implemented in Python3. Test your program using LeetCode's testing mechanism.

20. Search a 2D Matrix

21. Search a 2D Matrix II