- See Practice Problems Set 1 for instructions.

When analyzing the running time of an algorithm you may make reasonable assumptions about how various operations might be implemented in a computer. In particular, you may assume that the following take constant time:

- Arithmetic operations involving two numbers, such as addition, subtraction, multiplication, and division with or without remainder.
- Comparisons involving two numbers, such as testing if $x < y$, $x = y$, or $x > y$.
- Assignments of values to variables.
- Accessing an array element given its index.

You must explicitly state all *non-trivial* assumptions that you make. If in doubt, over-communicate.

1. Derive an upper bound on the running time of procedure EUCLID(x, y) given below in terms of the *total* bit size n of its two arguments. That is, n denotes the sum of the counts of bits of x and y. Your final answer should be a polynomial in n.

```
DIVIDE(x, y)
 1  if x = 0
 2       q = 0
 3       r = 0
 4       return (q, r)
 5  z = ⌊x/2⌋
 6  (q, r) = DIVIDE(z, y)
 7  q = 2 × q
 8  r = 2 × r
 9  if x is odd
10       r = r + 1
11  if r ≥ y
12       r = r − y
13       q = q + 1
14  return (q, r)
```

```
EUCLID(x, y)
1  if y = 0
2       return x
3  (q, r) = DIVIDE(x, y)
4  gcd = EUCLID(y, r)
5  return gcd
```

2. Prove the following statement about finite, simple, undirected graphs G by induction on $|V|$, the number of vertices in G.

Claim

Any connected graph $G = (V, E)$ that satisfies $|V| = |E| + 1$, is acyclic.

3. An *isolated* vertex in a graph is a vertex that has no edges incident with it. Consider the following claim and its proof, about finite, simple, undirected graphs[1].

Claim

Any graph $G = (V, E)$ that has at least two vertices, and with $|V| = |E| + 1$ and no isolated vertices, is acyclic.

Proof. By induction on $|V|$:

- Base case: $|V| = 2, |E| = 1$: correct by inspection.
- Inductive assumption: the claim holds for all graphs with at most k vertices, for some $k \geq 2$.
- Inductive step: Let $G = (V, E)$ be a graph with $|V| = k = |E| + 1$, which has no isolated vertices. Consider an arbitrary graph G' obtained from G by adding (i) a new vertex v, and (ii) an arbitrary edge from v to some vertex of G. Then G' (i) has $k+1$ vertices and k edges, (ii) has no isolated vertices, and (iii) is acyclic. Hence proved.

□

   (a) Come up with a *counterexample* for this claim.

   (b) What is wrong with the above proof?

4. Go through the Wikipedia page on sorting algorithms.

5. Recall the "small step" problem that we discussed in class, as part of solving the sorting problem:

[1] Look up these terms if you need.

Small Step

Input: An array $A[0, 1, \ldots, (n-1)]$ of n integers, and an index $0 \leq i \leq (n-1)$ such that the part $A[0, 1, \ldots, (i-1)]$ is sorted.

Output: An array $B[0, 1, \ldots, (n-1)]$ where

- $B[0, 1, \ldots, i]$ is a sorted version of $A[0, 1, \ldots, i]$
- $B[(i+1), \ldots, (n-1)]$ is identical to $A[(i+1), \ldots, (n-1)]$

Try to come up, on your own, with the pseudocode for a procedure SmallStep(A, i) that solves this problem. The code that you come up with does *not* have to be the same as what we saw in class; it just has to correctly do the job.

More generally: the idea is not *to memorize pseudocode like a poem(!!), but to be able to derive it from reasoning when needed.*

The next couple of questions are based on the following pseudocode for Insertion-Sort that we saw in class.

Insertion-Sort(A)

```
1  i = 1
2  while i ≤ (n − 1)
3        next = A[i]
4        j = (i − 1)
5        while j ≥ 0 and A[j] > next
6              A[j + 1] = A[j]
7              j = (j − 1)
8        A[j + 1] = next
9        i = i + 1
10 return A
```

6. Translate this pseudocode to Python. Using a driver program, verify that your code correctly sorts a sufficiently large number of randomly generated lists integers, of various lengths.

7. Come up with a set of useful invariants for the **while** loop in lines 5–7. The notion of usefulness here is that the set of invariants should help us formally complete the proof of correctness of the invariant for the *outer* **while** loop that we saw in class. Show that your invariants are indeed invariant, and useful.

8. There are usually various notions of what a "partially solved instance" looks like, for any given problem. Here is a different such notion for the sorting problem, than the one we saw in class: The original task is to sort an array $A[0, 1, \ldots, (n-1)]$ of n integers in non-decreasing order. The partially solved instance is a pair (B, i) where B is an integer array with the following properties:

   - B contains exactly the same multiset of elements as A.
   - The prefix $B[0, 1, \ldots, (i-1)]$ of the first i elements of B, are the smallest (with repetitions, if any) i elements of A, in non-decreasing order.

   (a) How is this different from the partial solution that we saw in class for the sorting problem?

   (b) What is a notion of making a small bit of progress from this partial solution towards a complete solution?

   (c) Write the pseudocode for an algorithm that makes this small bit of progress.

   (d) Write the complete pseudocode for an algorithm that solves the sorting problem by bootstrapping the small-step algorithm from the previous step.

   (e) Argue that the algorithm for sorting that you came up with in this manner, correctly sorts valid inputs. Use loop invariants as needed.

9. Translate the pseudocode from your solution to Question 8 to Python. Using a driver program, verify that your code correctly sorts a sufficiently large number of randomly generated lists integers, of various lengths.

10. Read up on the Binary Search algorithm, and understand why it takes at most $c \log_2 n$ steps on a (sorted!) input array with n elements.

11. All arrays in this problem are indexed starting from 0. The input consists of two arrays A, B of integers, each of which is sorted in increasing order. Array A contains n integers, and array B contains $n+1$ integers. No integer repeats in either array, or across the two arrays; that is, each of A, B is in fact a set, and the list of all the elements in A or B, is also a set.

    Let C be the sorted—in increasing order—array that is obtained by merging arrays A and B into one array. Note that C is *not* given as part of the input; we define it just to make it easier to describe the problem.

    (a) What is the index of the median element of array C? That is, what is the value of i such that $C[i]$ is the median element of C? Why is this i unique? Justify your answer.

(b) Write the pseudocode for an algorithm that takes A, B as input, takes at most cn steps for some constant c, and finds the median element of the array C defined as above. Note that C is *not* part of the input to this problem.

Argue that your algorithm is correct, and that it runs within the specified number of steps. What is the constant c that you get?

(c) It turns out that since the input is sorted, we can devise an algorithm that solves the problem much faster. For this we will use the idea that we saw in class, namely: solve one or more simpler problems, and use their solutions to solve the main problem.

i. **Sub-problem 1**: The input to this sub-problem consists of arrays A, B and an index i into array A, and a *guarantee* that $A[i]$ is the last element of A that appears *before* the median element in the (*unknown, and uncomputed*) array C. The required output is the median element of C.
Note that the guarantee is *not* that $A[i]$ is the element of C that comes just before the median element of C. If you don't see why this is the case: read the problem statement again, a few times till you get it. If you have trouble even after trying for some time: ask others and figure this out.
Write the pseudocode for an algorithm that solves this sub-problem using a *constant* number of steps in the worst case. Note that this algorithm does *not* have access to C. Argue that your algorithm is correct, and that it runs within the specified number of steps.

ii. **Sub-problem 2** The input to this sub-problem consists of arrays A, B and an index i into array A, and the output is *whether* $A[i]$ is the last element of A that appears *before* the median element in the (*unknown, and uncomputed*) array C. The output is thus either "Yes" or "No".
(As in the previous sub-problem: the question is *not* whether $A[i]$ is the element of C that comes just before the median element of C.)
Write the pseudocode for an algorithm that solves this sub-problem using a *constant* number of steps in the worst case. Note that this algorithm does *not* have access to C. Argue that your algorithm is correct, and that it runs within the specified number of steps.

iii. "Bootstrap" the solution to **Sub-problem 2** to design an algorithm that takes A, B as input, takes at most $c \log n$ steps for some constant c, and finds the median element of the array C defined as above. For this, modify the Binary Search algorithm to find an index i into A for which **Sub-problem 2** returns "Yes".
Once again, note that C is *not* part of the input to this problem.

iv. As a final step, think of various extreme/corner cases of input that might

not have been covered by the above algorithm. For each such case, write down the pseudocode for an algorithm that solves the problem in a constant number of steps.

v. Combine all these to arrive at an algorithm that solves the problem of finding the median of array C in at most $c \log n$ steps for some constant c. Argue that your algorithm is correct, and that it runs within the specified number of steps.

12. Prove that when given a natural number n as argument, the following procedure prints "Hi there!" at most $cn \log_2 n$ times for some constant c which is independent of n.

```
HiThere(n)
1  for i := 1 ; i ≤ n ; i := (i + 1)
2        for j := i ; j ≤ n ; j := (j + i)
3              print("Hi there!")
```

13. (a) Design an algorithm which

    1. Takes a positive integer n as its input
    2. Produces as its output an array $LPD[0, 1, \ldots, n]$ where $LPD[0] = 0$, $LPD[1] = 1$, and for each $2 \leq i \leq n$, $LPD[i] = p$ where p is the smallest prime divisor of i.
       - "LPD" stands for **L**east **P**rime **D**ivisor
       - Write this algorithm as a procedure LeastDivisors(n) that takes n as an argument and returns the array LPD.
       - Example: LeastDivisors(10) should return the array $LPD = [0, 1, 2, 3, 2, 5, 2, 7, 2, 3, 2]$.
    3. Takes at most $cn \log_2 n$ computer steps ("time") for some constant c which is independent of n.
       - You may assume that assigning a number to a variable, adding two numbers, and accessing a single array element given its index, can each be done in a constant number of computer steps. You may *not* assume that Euclidean division of one number by another can be done in a constant number of computer steps.

    Hint: See if you can adapt the loops in HiThere *from Question 12 to make this happen.*

    (b) Prove that your algorithm correctly computes the required output.

(c) Prove that it does so within the required running time bound, given the various assumptions.

14. (a) Design an algorithm which

    1. Takes a positive integer n and the array $LPD[0, 1, \ldots, n]$ described in Question 13 as its inputs
       - That is: we assume here that somebody has computed $LPD[0, 1, \ldots, n]$ and given it to us "for free".
    2. Produces as its output the prime factorization of n
       - What would be a good way to *represent* this output? You may choose any one such way to represent the output.

    (b) Prove that your algorithm correctly computes the required output.

15. Procedure ArraySearch below is a student's attempt at expressing the binary search algorithm in pseudocode. The input to ArraySearch consists of

    1. An array $arr[0, 1, \ldots, (n-1)]$ of n integers sorted in non-decreasing order,
    2. An integer val, and,
    3. Two integers $0 \leq begin \leq end \leq (n-1)$.

    (a) Check if the pseudocode works correctly as a search algorithm. That is:

    - *Either* prove that ArraySearch, when invoked as ArraySearch$(arr, val, 0, (n-1))$ with valid inputs as described above
      - Returns an index i such that $arr[i] == val$, if val is present in arr, and
      - Returns -1 (which is an invalid index for arr) if val is not present in arr
    - *Or* come up with a valid input on which ArraySearch fails to do the above task.

    (b) Show that—irrespective of whether it is a correct search algorithm—ArraySearch returns in at most $c \log_2 n$ steps, for some constant c that is independent of n, when the number of elements in its first argument arr is n.

```
ArraySearch(arr, val, begin, end)
1  if begin ≤ end
2       mid = ⌊(begin + end)/2⌋
3       if arr[mid] == val
4            return mid
5       if arr[mid] > val
6            return ArraySearch(arr, val, begin, (mid − 1))
7       if arr[mid] < val
8            return ArraySearch(arr, val, (mid + 1), end)
9  return −1
```

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

16. ★ Add two numbers

17. ★ Longest Substring Without Repeating Characters

18. ★ Longest Palindromic Substring

19. ★ Reverse Integer