CS 330, Fall 2020, Homework 1
Due Wednesday, September 9, 2020, 11:59 pm EST, via Gradescope

Homework Guidelines

Collaboration policy Collaboration on homework problems, with the exception of programming assignments and reading quizzes, is permitted, but not encouraged. If you choose to collaborate on some problems, you are allowed to discuss each problem with at most 5 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (these include anyone not enrolled in the class) is strictly forbidden.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

Solution guidelines For problems that require you to provide an algorithm, you must give the following:

1. a precise description of the algorithm in English and, if helpful, pseudocode,
2. a proof of correctness,
3. an analysis of running time and space.

You may use algorithms from class as subroutines. You may also use any facts that we proved in class.

You should be as clear and concise as possible in your write-up of solutions. A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand. Points might be subtracted for illegible handwriting and for solutions that are too long. Incorrect solutions will get from 0 to 30% of the grade, depending on how far they are from a working solution. Correct solutions with possibly minor flaws will get 70 to 100%, depending on the flaws and clarity of the write up.

1. **Array operations and asymptotic running times** (10 points)

(a) Consider the following pseudocode:

Algorithm 1: TestAlg(A)

Input: A is an array of real numbers, indexed from 1 to n

```
1 n ← length(A) ;
2 for j ← 1 to ⌊n/2⌋ do
3 |   k = n + 1 − j;
4 |   A[j] ← A[j] + A[k] ;
5 |   A[k] ← A[j] − A[k] ;
6 |   A[j] ← A[j] − A[k] ;
```

Which of the following statements are true at the end of every iteration of the **for** loop?

i. The sub-array $A[1 \dots j]$ contains its original contents in their original order

ii. The sub-array $A[1 \dots j]$ contains the original contents of sub-array $A[(n+1-j) \dots n]$ in reverse order.

iii. The sub-array $A[1 \dots j]$ contains its original contents in reverse order.

iv. The sub-array $A[(n+1-j) \dots n]$ contains its original contents in their original order.

v. The sub-array $A[(n+1-j) \dots n]$ contains the original contents of sub-array $A[1 \dots j]$ in reverse order.

vi. The sub-array $A[(n+1-j) \dots n]$ contains its original contents in reverse order.

For the ones that you select as always true, write a one sentence justification.

ANSWER

Out of the statements above, the true ones are [ii] and [v]. Both for the same reason, as what the algorithm does is invert the order of the array of real numbers.

This occurs as the algorithm grabs the two elements at the same distance from the center: A[j]=X and A[k]=Y, then adds them and puts them into A[j] (A[j]=X+Y), then subtracts A[k] from A[j] and puts it into A[k] (A[k]=X), and then subtracts A[k] from A[j] again and puts it into A[j] (A[j]=Y).

After repeating this for every element until the center, we get an inversed array.

Aside: Statements that hold on every iteration of a loop, like the correct options above, are called *loop invariants*. We will use them a lot when we analyze algorithms.

(b) Consider the following *SwapAlg*( ) algorithm.

Algorithm 2: SwapAlg(A)

Input: A is an array containing the first n positive integers. It is indexed 1 to n.

```
1 n ← length(A) ;
2 swaps ← 0;
3 for i = 1 to n do
4     for j = 1 to n − i do
5         if A[j] > A[j + 1] then
6             A[j] = A[j] + A[j + 1];
7             A[j + 1] = A[j] − A[j + 1];
8             A[j] = A[j] − A[j + 1];
9             swaps + +;
```

Output: *swaps*

i. Give the best asymptotic upper bound you can on the running time of *SwapAlg*( ) as a function of n using big-Oh notation. Explain your computation.

ANSWER
The Big-Oh upper bound is O(2n), this is easily calculated if you take a couple examples in worst case (the array is inverted) as, given an array that is sorted biggest to smallest, swaps actually returns the number of iterations.
For an array of size six sorted as such, swaps is 15, for an array of size 5 it's 10, etc... from these it's easy to derive that swaps, in these worst case-scenarios, will always be 2n+b, so if we discard the b, we get O(2n).

ii. Explain in one or two sentences what value the variable *swaps* is tracking. What is the number that this algorithm returns?

ANSWER
Swaps is tracking the amount of times a number shifts in the array in order to sort it. So in a sorted array from smallest to largest swaps will be 0, and in a sorted array from largest to smallest, swaps will be the runtime.

(c) Consider the *decAlg*( ) algorithm.

Algorithm 3: decAlg(A)

Input: A is an array of integers. It is indexed 1 to n.

```
1  n ← length(A) ;
2  dec ← 0;
3  for i = 2 to n do
4      j ← i − 1;
5      while j > 1 AND A[j − 1] > A[j] do
6          temp ← A[j − 1];
7          A[j − 1] ← A[j];
8          A[j] ← temp;
9          j − −;
10         dec + +;
```

Output: dec

i. Give the best asymptotic upper bound you can on the running time of *decAlg*( ) as a function of n using big-Oh notation. Explain your computation.

ANSWER
This algorithm, just as the one above, has a runtime of O(2n). Despite starting one item ahead of what is seen in the algorithm before (i=2), it will still perform swaps starting in i-1, so it ends up having the same upper bound as swapAlg.

ii. Observe, that both in *SwapAlg*( ) and in *decAlg*( ) pairs of values in A are swapped. Both variables *swap* and *dec* keep track of the number of swaps performed in their respective algorithms. State and give an exact proof of what the relationship between the two variables are when the algorithms are run on the same input A. (The answer we are looking for is that *swap* is always smaller/equal/larger than *dec* or that it varies depending on the input array.)(*Hint: One way of proving it is to show that for any two values – initially at index A[x] and A[y] – if they are swapped at any given time in swapAlg( ) then they eventually will be swapped in decAlg( ) and vice verse.*)

ANSWER
Given the same input array, *swap* and *dec* will always be the same, as each will count the amount of times the items inside that array have been swapped so to sort it.
For instance, if we take two values A[x]=G and A[y]=H, if in swapAlg A[y] is switched with A[x] that means the items were out of order by a certain amount, which would be added into *swap*, this means that eventually these two items would have to be switched in order for decAlg to sort them, as for the amount of times, we can derive from the fact that both of these algorithms have the same runtime (ie: they end up having the same cycles), then the amount of times that A[x] and A[y] are switched in decAlg is the same as swapAlg, so the amount added to *dec* is the same as that added to *swap*.

(d) Consider the *CountAlg*( ) algorithm.

Algorithm 4: CountAlg(A)

Input: A is an array containing the first n positive integers. It is indexed 1 to n.

```
1 n ← length(A) ;
2 count ← 0;
3 for i = 1 to n do
4 |   for j = i + 1 to n do
5 |   |   if i < j AND A[i] > A[j] then
6 |   |   |   count + +
```

Output: *count*

i. Give the best asymptotic upper bound you can on the running time of *countAlg*( ) as a function of n using big-Oh notation. Explain your computation.

ANSWER
Just as in the previous two algorithms, count will return the amount of times a number would need to be moved in order to sort the array, except this time the algorithm doesn't actually sort the array, only iterates through it, which would theoretically speaking shorten the runtime, albeit by a negligible amount.
With that said, the algorithm would still perform in O(2n), as for each item it would run n(i+1), so for an array of size 6 it would run 15 times (or 2n+3) and for an array of size 5 it would run 10 times (2n).

ii. In fact, *count* in *countAlg*( ) and *swap* in *SwapAlg*( ) are closely related to each other. This is a non-trivial relationship that we're going to cover in detail later this semester. For now, give your best guess based on your intuition of what this relationship might be. Make sure you clearly state what your observation is and provide some justification. For the latter, show us how you came up with the observation. (e.g. by showing your computation for the two values on some specific examples) (*Note: tracing algorithms on specific examples is very helpful in understanding the algorithm. But you need to do it by yourself to truly benefit from this.*)

ANSWER
Unlike swapAlg and decAlg, swapAlg and countAlg share the same basic structure of iterations, except instead of iterating through every item and then through every other item except its equivalent in the other half of the array (and those after), countAlg iterates through every item and then through every item except those who come before it.

This means they are basically doing the same thing, except grabbing different ends of the array. Meaning they will output the same value for the same Array input, as well as have the same runtime.

2. **Group assignments** (10 points)

In CS330, the instructors decide to divide the students in to small groups that will be assigned projects. In order to encourage students to get to know their classmates, no two students

living on the same floor in the dorms can be assigned to the same group. (For simplicity assume that every student resides on campus.) The size of the individual groups doesn't matter, however the instructors want to create as few groups as possible.

Denote the total number of students by n and the number of floors by k.

(a) True or False? the number of groups one needs is always at least the maximum number of students living on any given floor. Justify your answer by giving a one-line proof (if it's true) or a counterexample (if it's false).

ANSWER
The statement above is True, as if there were any less groups than the maximum number of students on any given floor, then at least 1 of the students of said floor would need to share a group with another student from the same floor.

(b) Describe a simple algorithm, $Floors(A)$, that takes an array A as its input, such that $A[i]$ contains the floor ID for student i. The output is an array or hash table (dictionary in Python) $floor$, such that $floor[j]$ consists of the list of students living on floor j. Write concise and clear *pseudocode* for your algorithm. (*Note: make sure you clearly indicate in your code what data structure(s) you are using*) Give a one sentence explanation on how your algorithm works.

ANSWER
Note: floor is a dictionary with floorID keys and int array values. Student ids are indicated as indexes of A, so they are ints, and floorIDs are also ints. example[i].append(j) appends the value j to a dictionary example under key i.
This algorithm grabs the input array, creates the dictionary, and iterates through the input array adding each student to their respective floor.

Algorithm 5: Floors(A)

Input: A is an array of students where $A[i]$ indicates the floor id of student i.

```
1 n ← length(A) ;
2 floor ← {};
3 for i = 1 to n do
4 |   f ← A[i];
5 |   floor[f] ← floor[f].append(i)
```

Output: $floor$

(c) Find a *simple* algorithm, $AssignStudents(A)$, that takes an array A as its input, such that $A[i]$ contains the floor ID for student i. The output is data structure (e.g. array, hash table) B, where $B[j]$ contains a *list* of students assigned to group j. Write concise and clear *pseudocode* for your algorithm. (You may want to call the $Floors(A)$ function as a subroutine in your algorithm – it's of course not required.) Give a short explanation in English on how your algorithm works.

ANSWER

Note: floors is a dictionary with int keys and int array values, with it's keys being the floorIDs, and the array values the studentIDs. On the other hand groups is a dictionary of size maxStudents, with it's keys being ints, and it's values being int arrays, where the keys are the groupIDs, and the values being studentID arrays. maxStudents is an int, and example[i].append(j) appends the value j to a dictionary example under key i.
This algorithm is distinctively divided into two parts, the first where it gets an organized Dictionary of students per floor, and the maximum number of students on any given floor. Then it creates a dictionary of groups and appends to each group the student in the same position on each floor, if they exist. This will create unheaven groups, but it will also assure no two students on the same floor are together.

Algorithm 6: AssignStudents(A)

Input: A is an array of students where $A[i]$ indicates the floor id of student i.

```
1  floors ← Floors(A);
2  maxStudents ← 0;
3  for i = 1 to length(floors) do
4      if length(floors[i]) > maxStudents then
5          maxStudents ← length(floors[i]);
6  groups ← new Dic(maxStudents);
7  for i = 1 to maxStudents do
8      for j = 1 to length(floors) do
9          if length(floors[j]) <= i then
10             groups[i] ← groups[i].append(floors[j][i])
```

Output: *groups*

(d) Give the best asymptotic upper bound you can on the running time of *AssignStudents*( ) as a function of the number of students, n, using big-Oh notation. Explain your computation.

ANSWER

The asymptotic upper bound of *AssignStudents*( ) as described above would not only be based on the number of students, but also the number of floors and the maximum students on any given floor. Given n as the number of students f the number of floors and m the maximum students on any given floor we have that big-Oh of *AssignStudents*( ) as $O(n+f*(m+1))$, it is harder to further abstract this without making assumptions as per what the maximum student on any given floor or the number of floors is, especially since this number could vary wildly.

Note: Your running time analysis will always be graded based on your ability to analyze *your* algorithm. That is, you have to give an analysis of the exact algorithm that you have written. That requires that you be specific about the data structure you use and what the running time of certain operations are on that structure. You may simply state your assumptions, as long as they are reasonable (e.g. you may reference what you

know from CS112 about each structure.) Do not forget to include the running time of the $Floors(A)$ subroutine if you're using it!