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

Student: Justin DiEmmanuele
Collaborators: Shilpen Patel, George Padavick, Matthew Gilgo

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.

Solutions will be in blue

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
   - FALSE

ii. The sub-array $A[1 \dots j]$ contains the original contents of sub-array $A[(n+1-j) \dots n]$ in reverse order.
   - TRUE
   - The algorithm swaps the contents of the first and last index, second and second to last index etc. This results in a mirrored array which would have the property described above.

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

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

v. The sub-array $A[(n+1-j) \dots n]$ contains the original contents of sub-array $A[1 \dots j]$ in reverse order.
   - TRUE
   - This is true for the same reasons part ii is true - the algorithm is mirroring $A[1 \dots j]$ and putting it in the position of $A[(n+1-j) \dots n]$ and vice versa

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

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

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.

- $O(n^2)$
- The program will run the outer loop n times and the inner loop $n-i$ times. The inner loop therefore runs 1 less time for each iteration of the outer loop. The run time will be something like: $(n-1)+(n-2)\ldots(n-n)$ having n terms. n terms with a function of n runs per term results in an n^2 run time. The run time will be lower than n^2 but still falls under the category of $O(n^2)$

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

- *swaps* is tracking the number of times two adjacent array values swap indexes. This algorithm returns this number - how many times two array values had their indexes swapped.

(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 + 1 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.

- $O\left(n^2\right)$
- This has the same time complexity as algorithm 2. It would have the same representation of the number of executions but in reverse order $(n-1)+(n-2)\ldots(n-n)$. Since there are n terms of functions of n iterations - the complexity is also $O(n^2)$

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.*)

- *swaps* and *dec* in *SwapAlg* ( ) and *decAlg* ( ) will be equal for any array passed to the algorithms.
- Both algorithms are iteratively comparing adjacent values and swapping them if they are not in increasing order. Both algorithms will compare every adjacent pair of index values.
- For both *SwapAlg* ( ) and *decAlg* ( ) the input A will have adjacent indexes checked for swap the same number of times. Therefore, the number of swaps will be the same even if they occur at different iterations of the loop.

(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.

- $O\left(n^2\right)$
- This algorithm has the same run time as algorithms c and d. The outer loop runs n times and the inner loop runs $n-i$ times where i is whichever number iteration the outer loop is on. This results in a number of executions like: $(n-1)+(n-2)\ldots(n-n)$. This is n terms that have n run time giving an upper bound of $O(n^2)$.

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.*)

- *countAlg* ( ) and *SwapAlg* ( ) are related in that they both return the number of swaps that would be performed if an array were to be sorted into ascending order.
- Both *countAlg* ( ) and *SwapAlg* ( ) end up comparing every combination of numbers in the array, *countAlg* ( ) just counts the instances where the lesser index is greater instead of swapping the values like *SwapAlg* ( ).

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).

- TRUE
- If there are less groups than numbers of students on any given floor it would be impossible to separate the kids on that floor into distinct groups. Lets say n kids one one floor were to be split into m groups. We would iterate through the kids assigning each to empty groups. If m $<$ n, we would come to the realization that there are no empty groups to assign the kid after m iterations.

(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.

Algorithm 5: Floors(A)

Input: A is an array containing the floor ID $A[i]$ for student i

```
n ← length(A) ;
f ← hash_table{} ;
for i = 1 to n do
    f[A[i]] ← [ ]
for j = 1 to n do
    f[A[j]].append(j)
```

Output: f

Explanation The Floors algorithm initializes a hash table or dictionary with the keys corresponding to each unique floor ID and its value an empty linked list. It then appends students IDs to their corresponding floor ID key.

(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.

Algorithm 6: AssignStudents(A)

Input: A is an array containing the floor ID $A[i]$ for student i

```
n ← length(A) ;
groups ← hash_table{} ;
max_students ← 0 ;
f ← Floors(A) ;
for key in f do
    num_students =← length(f[key]) ;
    if num_students > max_students then
        max_students = num_students
for i = 1 to max_students do
    groups[i] = [ ]
count ← 0 ;
for key in f do
    for student in f[key] do
        groups[count % max_students].append(student) ;
        count++
```

Output: *groups*

Explanation All this algorithm does is set the number of groups to the maximum number of students as that is the minimum groups needed per section a. This is also

the maximum number of groups needed as all other floors can be distributed in these groups without having a conflicting grouping. The algorithm then iterates through each floor of students and places them in groups of incrementing number. Since the number of groups is greater than or equal to the number of students in any floor, this will both satisfy the requirement of having no students in the same group from the same floor and give the minimum number of groups possible.

(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.

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!

Answer

- The variable assignments in the first 3 lines run in O(1) time
- Line 4 calls the Floors algorithm and assigns the output to the variable f. The Floors algorithm has a run time of $O\left(n^2\right)$ shown below:
  - Variable assignment $O\left(1\right)$
  - The first loop is $O\left(n\right)$ as the list initialization is $O\left(1\right)$ and runs n times
  - The second loop is $O\left(n^2\right)$ as accessing a dictionary or hash table is $O\left(n\right)$ and it runs n times
- Worst case for the loop starting on line 5 is $O\left(n^2\right)$. The loop itself in worst case would run n times if every student lives on different floors. The lookup in the hash table or dictionary is $O\left(n\right)$ and is run max n times in the loop. The comparison and assignment within are $O\left(1\right)$.
- The loop starting on line 9 will run a maximum of n times. The initialization of each list in the hash table groups is an $O\left(1\right)$ operation. This makes this loop as a whole $O\left(n\right)$
- The loop beginning on line 12 will execute maximum n times on the outer loop. The inner loop will execute n times as it is iterating through all the values loaded into f - which are the original values of array A with length n. The second loop has a lookup into a hash table which has a run time of $O\left(n\right)$. The lookup into the group hash table is $O\left(n\right)$ and the append method is $O\left(1\right)$. Finally the count increment is $O\left(1\right)$. This makes the total runtime of this group of loops $O\left(n^4\right)$.
- **The asymptotic upper bound is is $O\left(n^4\right)$ as the program is dominated by the loop in line 12.** To improve, there may be a way to move to using arrays that can be accessed in $O\left(1\right)$ time.