CIS 121 — Data Structures and Algorithms
Solutions to Homework Assignment 2
September 18, 2018

Important Note: Do not distribute or share these solutions. They are strictly for the personal use of students registered in the Fall 2018 edition of CIS 121 at the University of Pennsylvania. Any violations will be referred to the Office of Student Conduct.

1. [14pts - CIS 160 Review]
   a. Let \( G \) be a simple undirected graph with \( n \geq 2 \) vertices. Show that if \( G \) is not connected, then the sum of degrees of some pair of vertices in \( G \) is less than \( n - 1 \).

   b. You have a normal deck of 52 playing cards. Suppose you cut the deck once in a random position. Let \( X \) be the number of cards that start out in the lower half before the cut and remain in the lower half after the cut.
      i. What is \( E[X] \)? Justify your answer.
      ii. Now suppose you cut the deck 121 times in random positions. In that case, what is \( E[X] \)? Justify your answer.

* To cut a deck, one takes a contiguous portion of the deck off the top and places it beneath the rest of the deck.

Solution.
   a. Let \( u \) and \( v \) be some pair of vertices in \( G \) that belong to different connected components, say \( C_1 \) and \( C_2 \). Let \( n_1 \) and \( n_2 \) be the number of vertices in \( C_1 \) and \( C_2 \) respectively. We have
      \[
      \deg(u) + \deg(v) \leq (n_1 - 1) + (n_2 - 1) \leq n - 2
      \]
   b. For both parts (a) and (b), \( E[X] = 13 \).
      i. Each card is equally likely to move to the 52 spots. Hence, for each of the 26 cards in the lower half, there is a 1/2 chance of ending up in the lower half, so \( E[X] = 1/2 \cdot 26 = 13 \).
      ii. The answer is still 13 since whether you cut once or 121 times, each card is equally likely to end up in any given spot.

2. [14pts - CHOCOLATE!!!] Imagine you have an \( m \times n \) chocolate bar that you want to divide into individual \( 1 \times 1 \) pieces. You may assume \( m \) and \( n \) are both positive integers. Suppose some algorithm \( A \) makes a series of “breaks” such that at the end of the algorithm, you are left with \( mn \) individual pieces. Note \( A \) can only “break” one piece at a time.
   a. Give a lower bound on the number of “breaks” \( A \) makes. Justify your bound.
   b. Give an algorithm and analyze its runtime to show that this lower bound is achievable, and thus is tight.
Solution.

a. The lower bound on the number of “breaks” a makes is $mn - 1$. Each time we perform a “break”, we increase the total amount of pieces by 1. Because we start with 1 piece (the entire chocolate bar) and end with $mn$ pieces, we must perform at least $mn - 1$ breaks in any correct algorithm.

b. Algorithm: Begin by breaking the chocolate bar between every row. Then, for each row piece, break it between every column.

Runtime: There are $m$ rows, so breaking the bar between every row takes requires $m - 1$ “breaks”. There are $n$ columns, so breaking a particular row between every column takes $n - 1$ “breaks”–because there are $m$ rows, this step will take $m(n - 1)$ “breaks” in total. Therefore, the entire algorithm will require $(m - 1) + m(n - 1) = mn - 1$ “breaks”.

3. [15pts - Big-Oh] Prove or disprove the following. In case of a proof, use the definitions of $O$, $\Omega$, $\Theta$ and give values of the constants in the definitions for which the conditions in the definition hold.

a. $5n\sqrt{n} = O(\frac{1}{4}n^2 - 10)$

b. $2^{5\lg n + \lg \lg n} \lg(n^5) = O(4^{3\lg n})$

c. $f(n)^2 = O(g(n)^2)$ given two functions $f$ and $g$ such that $f(n) = O(g(n))$.

Solution.

a. True. We will prove the claim by showing that there exist positive constants $c$ and $n_0$ such that

$$5n\sqrt{n} \leq c(\frac{n^2}{2} - 10), \forall n \geq n_0$$

Multiply both sides of the above inequality by 2 and rearranging the terms yields $cn^2 - 10n\sqrt{n} \geq 20c$ or, equivalently, $n\sqrt{n}(c\sqrt{n} - 10) \geq 20c$. One way to achieve our goal is by making sure that both $c\sqrt{n} - 10 \geq 1$, that is, $c\sqrt{n} \geq 11$, and $n\sqrt{n} \geq 20c$. The first inequality is satisfied if $c = 11$ and $n \geq 1$. Putting $c = 11$ in the second inequality we get $n\sqrt{n} \geq 220$ which is surely satisfied by $n \geq 220$. Since $220 > 1$ the first inequality is also satisfied. From all this it follows that taking $n_0 = 220$ and $c = 11$ we obtain what we wanted to show.

b. True. The left hand side simplifies to

$$(2^{\lg n})^5 \cdot 2^{\lg \lg n} \cdot (5 \lg n) = 5n^5 \cdot \lg^2 n$$

The right hand side simplifies to

$$(2^{\lg n})^6 = n^6$$

We will now prove the claim by showing that there exist constants $c$ and $n_0$ such that for all $n \geq n_0$, $5n^5 \lg^2 n \leq cn^6$. To show this it is sufficient to satisfy $5\lg^2 n \leq cn$. Note that we can set $c = 5$ and set $n_0$ accordingly. One such value is $n_0 = 2^5$. 
c. True. Since \( f(n) = O(g(n)) \), \( f(n) \leq c \cdot g(n) \) for some constant \( c \) and for all \( n \geq n_0 > 0 \). This means that \( f(n)^2 \leq c^2 g(n)^2 \) for all \( n \geq n_0 \).

4. [15pts - Recurrence Relations] For all recurrences below, assume that \( T(n) = 1 \) for \( n \leq 10 \) and \( n \) is an exact power of 2. Use the method of substitution to...

a. Prove or disprove that the solution to the recurrence \( T(n) = \frac{T(n^2)}{2} + \log n \) is \( \Theta(\log^2 n) \).

b. Solve the recurrence, giving your answer in \( \Theta \) notation: \( T(n) = \sum_{i=1}^{k} T\left(\frac{n}{a_i}\right) + n \) where \( a_1, a_2, \ldots, a_k \) are positive constants such that \( \sum_{i=1}^{k} \frac{1}{a_i} \leq \frac{99}{100} \)

Solution.

a. We first show that \( T(n) = \Omega(\log^2 n) \). Assume inductively that for some \( c > 0 \), \( T(n') \geq c \log^2 n' \) for all \( n' < n \), then we have that

\[
T(n) \geq c \log^2(n/2) + \log n \\
\geq c \log^2 n - 2c \log n + c + \log n
\]

Now, \( (c \log^2 n - 2c \log n + c + \log n) \) is at least \( c \log^2 n \) as long as we choose any \( c \in (0, \frac{1}{2}] \). The upper bound proof is symmetric, allowing us to conclude \( T(n) = \Theta(\log^2 n) \).

b. Clearly, \( T(n) = \Omega(n) \). We will prove using the substitution method that \( T(n) = O(n) \). This suffices to conclude that \( T(n) = \Theta(n) \) as \( T(n) \geq n \) by the definition of the recurrence.

Assume inductively that \( T(n') \leq cn' \) for all \( n' < n \). The base case for \( n = 0 \) is trivial. For \( n > 0 \),

\[
T(n) = \sum_{i=1}^{k} T\left(\frac{n}{a_i}\right) + n \leq \sum_{i=1}^{k} c\left(\frac{n}{a_i}\right) + n \leq c\left(\frac{99}{100}\right)n + n \leq cn
\]

for any choice of \( c \geq 100 \).

5. [16pts - Last Minute Complications] For each of the subproblems problem below, a clear description of your algorithm and an analysis of its time complexity is sufficient. No proof of correctness is needed.

a. Caroline is throwing a CIS 121 sleepover and wants to give out onesies to all the TAs. The onesies are sized by height, and Caroline realized last minute she only bought one onesie of each size. Therefore, she needs to make sure that she only has at most one TA of any given height. The TAs line up in two lines and order themselves in increasing order of height. Now, help Caroline find any TAs that have the same height as another TA and kick all but one of them (doesn’t matter who) out of the sleepover. Your algorithm should run in \( O(n) \) time, where \( n \) is the total number of TAs.
b. After dropping out of Penn, Shirali decides to open up her own comedy club restaurant. She maintained a schedule of performers, and performers were ordered by what time they planned to arrive at the restaurant (so they wouldn’t have to wait for too long). She initially had \( n \) comedians on the schedule ready to perform, but at the last minute, \( k \) of the comedians dropped out leaving her with \( n - k \) performers. Luckily, Shirali was able to find \( k \) relatively funny friends to replace them, but they were all arriving at the restaurant at different times. Given the original sorted schedule with \( n - k \) comedians on it, and given the list of \( k \) new comedians’ arrival times, design an \( O(k \log k + n) \) algorithm that creates a lineup sorted based on each comedian’s arrival time. You may assume \( 0 < k \leq n \), and that arrival times are distinct.

Solution.

a. Algorithm: We have two lines of people, each of which are sorted by increasing order of height. We will merge these two sorted lists of people into one list using the Merge algorithm used in MergeSort. While merging, however, we will keep a variable that stores the height of the last person added to the merged list. Each time we are adding the next person to the merged list, we check if the next person’s height is equal to the height in our stored variable. If they are equal, we won’t add the next person to the merged list. At the end of the algorithm, we will have a sorted list of TAs at the sleepover, where no two TAs who have the same height, so we can give each TA in the final list a onesie that fits them.

Runtime: The modified Merge algorithm takes \( O(n) \) time, since we are looking at each TA exactly once and adding them to the merged list. Maintaining the variable of the last person added to the merged list takes constant time, so this does not increase the \( O(n) \) runtime of Merge.

b. Algorithm: First, sort the \( k \) replacement comedians by their arrival times using Mergesort. Now that you have these \( k \) new comedians in sorted order, and the \( n - k \) other comedians already in sorted order, we can merge the two using the Merge algorithm discussed in lecture. As a result, we have the \( n \) comedians in sorted order by arrival time, and we can output this as the proper lineup.

Runtime: Sorting the \( k \) new comedians using Mergesort takes \( O(k \log k) \) time. Merging these \( k \) comedians with the remaining \( n - k \) comedians takes \( O(n - k + k) = O(n) \) time. However, since we don’t know whether \( n > k \log k \), we leave our final runtime as \( O(k \log k + n) \).

6. [11pts - Quicksort]

a. Array \( A = [3, 0, 2, 4, 5, 8, 7, 6, 9] \) has just been partitioned by the first step of the Quicksort algorithm. Which of \( A \)’s elements could have been the pivot? List all possible pivots if more than one exists. Give a brief justification (1-2 sentences max) for your answer.

b. For each sub-problem below, say how many recursive calls to Quicksort will be performed for each case (count the first one as well). You may assume that the
algorithm is selecting the pivot using: \( \text{Pivot} = A[(\text{Lo} + \text{Hi}) / 2] \) and that the arrays are 0-indexed. Inputs to Quicksort are:

(i) \( A = [1, 3, 2, 4, 5, 7, 6, 8, 9] \)
(ii) \( B = [2, 6, 9, 5, 1, 3, 4, 7, 8] \)
(iii) \( C = [7, 6, 8, 4, 9, 3, 5, 2, 1] \)

c. Please show the state of the array \( A \) in (i) after each partition when Quicksort is run on it.

Solution.

a. 4, 5, and 9. The pivot can be any element such that everything to the left is smaller and everything to the right is bigger, and these 3 elements fit that condition.

b. Answer to all three was 9 Quicksort calls. Justification for all three parts is on the last page. The bolded number is the pivot.

c. See figure on the last page.

7. [15pts - Sarah & Co] After graduating from Penn, Sarah decided to pursue her dream of becoming a famous arbiter of fashion, and opened a high fashion clothing boutique – Sarah & Co – located on 5th Avenue, right next to Saks’ flagship store. Designers from all over the world aspired to have their latest lines sold at Sarah & Co, and Sarah & Co quickly became THE store to discover and take part in all the latest fashion trends. Sarah’s clients include many celebrities and high-profile socialites, and she realized quickly that in order to maintain Sarah & Co’s high status, there were several services she needed to provide for her sellers and clients:

1. addItem() Sarah needs to be able to add new items to Sarah & Co’s collection. Items come in one at time.

2. sellLatestItem() Sarah noticed that many of her customers entered the store specifically looking to buy into the latest fashion trend. To address this, Sarah decides that any customer that walks in has to buy the most recent item added to the collection.

3. showOffMostExpensiveItem() To “wow” her high profile clients, Sarah wants to be able to quickly show off the most expensive item currently for sale in her store. Note: she doesn’t necessarily want to remove it from her store, but just be able to show it off.

Sarah’s clients don’t want to be kept waiting, so these three operations all must be done in \( O(1) \) time. How would you advise Sarah in how she should structure her store, in order to maintain all of these operations? Please prove the correctness and runtime for all operations.
Solution. Algorithm: We begin with two empty stacks, $S_1$ and $S_2$, where $S_1$ is the stack where the store items go in the order they arrive at the store and $S_2$ is an auxiliary stack such that the most expensive item seen so far is at the top.

More specifically, for each item, $i$, that arrives at the store, implementing `addItem` involves pushing it to $S_1$, comparing the price of $i$ with the item at the top of the stack $S_2$ and pushing $i$ to $S_2$ only if $i$ is at least as expensive as the current top item in $S_2$ or if $S_2$ is empty. Then `showOffMostExpensiveItem` just involves peeking at the top of $S_2$.

To implement `sellLatestItem()` just pop and return the top element in $S_1$ if it exists. If this element’s price is equal to the top of $S_2$, also pop the top-most element off $S_2$.

Correctness: The correctness of `addItem` follows because clearly the item is included in the stack after `addItem` was called. Similarly, `sellLatestItem` is correct since stacks are LIFO, so the popping the top of the stack is equivalent to returning the most recent item pushed onto the stack.

To prove the correctness of `showOffMostExpensiveItem`, it suffices to show that the top of $S_2$ is always the most expensive item seen so far that is still in the store.

We do this via induction on the number of operations performed:

IH: Assume that, for some $n$ and for every $k$, $1 \leq k \leq n$, after $k$ calls to `addItem`, `sellLatestItem`, and `showOffMostExpensiveItem`, the top of $S_2$ is the most expensive item seen so far and moreover, the number of elements in $S_2$ equal to some $x$ is the same as the number of elements in $S_1$ equal to $x$.

Base Case: The base case is when one operation has been performed. If this operation is not an `addItem` operation, then the claim holds vacuously since there are no elements in $S_2$ or $S_1$. So assume the first operation is an `addItem` operation. Once the first element is added to $S_1$ via `addItem`, it is immediately added to $S_2$ by construction. Since this element is trivially the maximum, and the same element is in both $S_1$ and $S_2$ exactly once, the base case holds.

IS: Consider a sequence of $n + 1$ operations that are either `addItem`, `sellLatestItem` or `showOffMostExpensiveItem`. Go back in time 1 step. By the induction hypothesis, we know at this time, after the $n$-th operation, the top of $S_2$ is the most expensive item seen so far. Now, we have three cases, based on what the $n + 1$-th operation is. Let $M$ be the item at the top of $S_2$ after the $n$-th operation.

(1) `addItem(x)`: `addItem` only adds $x$ to $S_2$ if $x \geq M$. Since the IH tells us $M$ was larger than everything in $S_1$, and $x \geq M$, that means $x$ is now larger than everything in $S_1$. Moreover, the number of occurrences of $x$ in both $S_1$ and $S_2$ increase by one. Since they were the same at iteration $n$, they must be the same at iteration $n + 1$. Hence the claim holds in this case.
(2) **sellLatestItem()**: We remove $M$ from $S_2$ iff the item $x$ popped from $S_1$ is equal to $M$. Suppose first that we do in fact remove $M$ once we remove $x$, i.e., $M = x$. The number of items in $S_1$ that are equal to $x$ and the number of items in $S_2$ equal to $x$ both decrease by exactly one, hence since these numbers were equal before the $n+1$-th operation, they are still equal afterwards. Now let $m$ be the element beneath $M$ in $S_2$. We still need to show that $m$ is at least as large as everything in $S_1$. Let $z$ be the largest element in $S_1$, and, seeking contradiction, suppose $z > m$. By the IH and the first part of the proof for this case, we know that the number of elements equal to $z$ in $S_1$ must be equal to the number of elements equal to $z$ in $S_2$. In particular, that means $z$ must be in $S_2$. But since elements in $S_2$ are increasing from bottom to top by construction (we only add an item to $S_2$ if the item is bigger than the previous top), this must mean that $z$ is at the top of $S_2$, since we assumed $z > m$. This clearly contradicts our assumption that $m$ was at the top of $S_2$. Hence the claim holds in this case.

If there was no such $m$, then $S_2$ is empty, so we must show that $S_1$ is also empty. If not, then that would mean that at time $n$, there was only one element on $S_2$ and multiple elements on $S_1$, which contradicts the IH.

Lastly, if we didn’t remove anything from $S_2$ during this operation, the the IH immediately implies that $M$ is bigger than everything in $S_1$ still (it was true before we removed an element from $S_1$ so it must certainly be true after) and that the number of elements equal to some particular $y$ in $S_2$ is the same as the number of elements equal to $y$ in $S_1$.

(3) **showOffMostExpensiveItem()**: This doesn’t add or remove any elements from $S_1$ or $S_2$. Hence the state of the stacks is exactly the same as it was after the first $n$ operations, so the claim holds by the IH.

Please note, if $S_2$ is empty after the $n$-th operation, then so is $S_1$. Hence this case reduces to the base case, which was already proved above.

*See justification for Question 6b and 6c on next page.*
Figure for 6i

Figure for 6ii

Figure for 6iii