Collaborators: Steve Li

Problem 1. We will prove that StoogeSort correctly sorts any list with $n \geq 1$ elements via induction.

Let us denote the list to be sorted as A. For our base cases, we have that when $n = 1$, A is already sorted, and when $n = 2$, StoogeSort compares $A[0]$ and $A[1]$ and swaps them if $A[0] > A[1]$, thus returning a sorted list.

Now, for any $n \geq 3$, suppose StoogeSort correctly sorts any list with less than n elements. We want to show that StoogeSort also correctly sorts a list with n elements.

Let us denote the first element of A as i and the last element as j. Then suppose $k = \left\lfloor \frac{j}{3} \right\rfloor$, and let A_1, A_2, A_3 denote to the three "thirds" of A as follows.

$$\begin{aligned} A_1 &= A[i \dots (i + k - 1)] \\ A_2 &= A[(i + k) \dots (j - k)] \\ A_3 &= A[(j - k + 1) \dots j] \end{aligned}$$

After the first phase, $A[i \dots (j - k)]$ (equivalently, $A_1 + A_2$) is sorted, which means that every element of A_2 is greater than or equal to every element of A_1; we denote this as $A_2 \geq A_1$. This also implies that $A[(i+k) \dots j]$ (or $A_2 + A_3$) contains at least length(A_2) elements that are greater than or equal to every element of A_1.

After the second phase, $A[(i + k) \dots j]$ (or $A_2 + A_3$) is sorted, which means that A_3 is sorted and that $A_3 \geq A_2$. We know that $A[(i + k) \dots j]$ has at least length(A_2) elements $\geq A_1$, and that length$(A_3) \leq$ length(A_2) because

$$\begin{aligned} j - (j - k + 1) + 1 &\leq (j - k) - (i + k) + 1 \\ k &\leq j - i - 2k + 1 \\ 3k &\leq j - i + 1 \\ 3 \left\lfloor \frac{j}{3} \right\rfloor &\leq j + 1 \text{ (since i = 0)} \end{aligned}$$

which implies that $A_3 \geq A_1$. It follows that $A_3 \geq A[i \dots (j - k)] = A_1 + A_2$.

After the third phase, $A[i \dots (j-k)]$ is again sorted. Since A_3 is also sorted, and $A_3 \geq A[i \dots (j-k)]$ (every element of A_3 is greater than or equal to every element of $A[i \dots (j - k)]$), this means that A is fully sorted, thus closing the induction.

Since StoogeSort performs constant-time comparison operations to sort size 1 or 2 lists, and then recursively calls itself on a size $\frac{2}{3}n$ list three times, we have the following recurrence:

$$T(n) = 3T\big(\frac{2}{3}n\big) + O(1)$$

Thus, by Master Theorem Case 1, we have $T(n) = \Theta(n^{\log_{3/2} 3}) = \Theta(n^{2.71})$.

Problem 2. (Part A)

1. We note that first few values of $T(n)$ are as follows.

n	$T(n)$
1	1
2	5
3	13
4	25
5	41

If we subtract 1 from each $T(n)$, then we get the triangular numbers times 4. Thus, we propose that the closed form of $T(n)$ is

$$T(n) = 2n(n-1) + 1$$

We prove this result using induction.

For the base case, $n = 1$, we have by the above equation that $T(1) = 2(1)(1-1) + 1 = 1$, which satisfies our initial condition, $T(1) = 1$.

Then suppose that for any $k >= 1$, $T(k) = 2k(k-1) + 1$ holds. We want to show that $T(k+1) = 2(k+1)(k) + 1$ also holds.

By the recurrence relation, we have

$$\begin{aligned} T(k+1) &= T(k) + 4(k+1) - 4 \\ &= [2k(k-1) + 1] + 4(k+1) - 4 \\ &= 2k^2 - 2k + 1 + 4k + 4 - 4 \\ &= 2k^2 + 2k + 1 \\ &= 2(k+1)k + 1 \end{aligned}$$

thus closing the induction.

2. We note that first few values of $T(n)$ are as follows.

n	$T(n)$
1	1
2	5
3	15
4	37
5	83

We propose that the closed form of $T(n)$ is

$$T(n) = 3 \times 2^n - 2n - 3$$

We will prove this result using induction.

For the base case, $n = 1$, we have by the above equation that $T(1) = 3(2^1) - 2(1) - 3 = 1$, which satisfies our initial condition, $T(1) = 1$.

Then suppose that for any $k >= 1$, $T(k) = 3 \times 2^k - 2k - 3$ holds. We want to show that $T(k+1) = 3 \times 2^{k+1} - 2(k+1) - 3$ also holds.

By the recurrence relation, we have

$$\begin{aligned} T(k+1) &= 2T(k) + 2(k+1) - 1 \\ &= 2[3(2^k) - 2k - 3] + 2k + 2 - 1 \\ &= 3(2^{k+1}) - 4k - 6 + 2k + 2 - 1 \\ &= 3(2^{k+1}) - 2k - 2 - 3 \\ &= 3(2^{k+1}) - 2(k+1) - 3 \end{aligned}$$

thus closing the induction.

(Part B)

1. $T(n) = \Theta(n^3)$

   (Master Theorem Case 3)

2. $T(n) = \Theta(n^{\log_4 17}) = \Theta(n^{2.044})$

   (Master Theorem Case 1)

3. $T(n) = \Theta(n^2 \log n)$

   (Master Theorem Case 2)

4. $T(n) = \Theta(\log^2 n)$

   We begin with $T(n) = T(n^{1/2}) + 1$.

   Let $m = \log_2 n$; then we have $T(2^m) = T(2^{m/2}) + 1$.

   Then let $R(m) = T(2^m)$; then we have $R(m) = R(\frac{m}{2}) + 1$. By Master Theorem Case 2, $R(m) = \Theta(\log m) = \Theta(\log^2 n)$.

Problem 3. Finding the longest path in a directed acyclic graph with real edge weights is equivalent to inverting (multiplying by -1) the weights of the edges and finding the shortest path in the inverted graph.

After we have inverted the graph, we use the topological sorting algorithm to represent the graph in topological order. From this topological order, we also initialize an array of length n such that each index corresponds to a node, and set each of the initial values to ∞. Then we iterate over the nodes in topological order, beginning with the starting source node, whose value we set to 0. We then proceed to the next node in topological order, and update the value in the array if the path to that node is less than the current value. In this fashion, we process each node in the graph, updating the shortest path value for each node by considering the shortest paths to its parents (which have been previously calculated) as well as the weights of the paths from the parents to the node, until we have found the shortest path for each node in the graph.

To then uninvert the paths, we multiply each shortest path value by -1 to obtain the longest path to each node in the original graph, and finally, we return the highest value - which is the longest path in the graph.

We will prove that this algorithm obtains the longest path for any directed acyclic graph with $n \geq 2$ nodes using induction.

The base case is when $n = 2$, in which case the algorithm will find the only possible path between the two nodes, which is also the longest path.

Now, for any $n \geq 3$, suppose that the algorithm correctly finds the longest path in any directed acyclic graph with less than n nodes. We will prove that the algorithm also correctly finds the longest path in any directed acyclic graph with n nodes.

Consider the directed acyclic graph without the nth node. We use our algorithm to determine the longest paths to each node in this graph from the source node. Since any parent of the nth node will be in this graph, we can find the longest path to the nth node by taking the maximum of all the longest path values of the nth node's parents and adding the weight of the corresponding edge to the nth node. Then we have an array of longest path values and return the highest value in the array as the longest path. Therefore, we have found the longest path in the directed acyclic graph with n nodes, thus closing the induction.

The time complexity of the topological sorting algorithm is $O(V + E)$. The algorithm itself visits each node (vertex) in the graph and loops over all parent nodes. Since the total number of such edges is $O(E)$, the time complexity of the algorithm (without topsort) is $O(V + E)$, and the overall time complexity (with topsort) is $O(V + E)$.

Problem 4. Let us assume that the layout of Sunnyvale is such that it can be represented by an undirected graph, where the streets of Sunnyvale correspond to the edges of the graph.

To traverse each edge (street) exactly once in each direction, we use a modified version of the depth-first search algorithm.

Unmodified DFS will visit each node in the graph and, in the process, traverse each tree edge twice (once forward, once backward). However, the remaining non-tree forward edges, back edges, and cross edges will not be traversed. To fix this, in the search function, we check not only if a neighboring node has been visited, but also if the edge to that node has been traversed, and if it has not yet been traversed, traverse it twice (forwards, then backwards to return to the current node), mark the edge as traversed, and continue with DFS.

This modified version of DFS will traverse each street of Sunnyvale exactly once in each direction (twice total) optimally, as it follows for the most part the DFS traversal.

We have proved in class that DFS correctly and optimally traverses (reaches all nodes) for graphs of any size n, and since this algorithm adds steps in the case of untraversed edges, but does not alter the DFS search itself, it will also correctly and optimally traverse (fully) graphs of any size n.

The time complexity of DFS is $O(V + E)$. At each node we additionally check if we have traversed each edge yet, which adds $O(E)$ steps, so the time complexity of our modified DFS algorithm is $O(V + E)$.

Problem 5. (Part 1) For this algorithm, we assume that each sorted list is a linked list. We begin by initializing a min-heap. Then, we insert the first element from each of the k sorted linked lists into the heap (which by definition heapifies after each insert). We then extract-min from the heap and add the extracted element to a new, "master" list. At this point, the next smallest element is either the new root of the min-heap or the next element in the list from which the extracted element came from, so we check if there is an element next to the extracted element in the linked list, and if so, we insert it into the heap. Regardless of whether there was another element in the linked list, we then extract-min again from the heap and append the extracted element to the master list, and repeat this process until all the original lists, as well as the min-heap, are empty, at which point we have a sorted master list.

The size of the heap is at most k elements at any given time, and since we perform n heapify operations (for n insertions), the time complexity of this algorithm is $O(n \log k)$.

(Part 2) We begin by initializing a min-heap. Since we know that the 1st number in sorted order is less than k positions away from the 1st position in the original list, we insert each of the first k numbers into the heap, thus guaranteeing that the 1st number in sorted order is now in the heap. We then extract-min from the heap to get the 1st number in sorted order and add it to a new "master" list. Next, we know that the 2nd number in sorted order is less than k positions away from the 2nd position in the original list, which implies that it was one of the first $k+1$ numbers in the original list. Since we have already inserted the first k numbers into the heap, we only need to insert the $k+1$th number into the heap, thus guaranteeing that the 2nd number in sorted order is now in the heap. We then extract-min again from the heap to get the 2nd number in sorted order and append it to the master list, and then repeat the process of adding the $k+j$th number from the original list to the heap (ensuring that the heap contains the jth number in sorted order), calling extract-min on the heap, and appending the extracted number to the master list, until the original list and the heap are both empty, at which point we have a sorted master list.

Since the size of the heap is at most k elements at any given time, and we perform n heapify operations, the time complexity of this algorithm is $O(n \log k)$.