1) Solution

Part 1: Assuming that the recursive sort sorts the 2/3 of the list correctly we can break this question into cases. Every number within the list should either be sorted into the beginning, B, the middle, M, or the end, E. Thus proving that each number arrives at their correct 1/3 section after the last sort will suffice to show that the algorithm works because we are assuming that the recursive sort works correctly. This can be shown in nine cases of where numbers could begin.

- L **in first third** If L begins in the first third of the list it will stay in the first third after the first sorting. Thus it will not be involved in the second sorting. In the third sorting it will remain in the first third where it belongs.
- L **in second third** If L begins in the second third of the list it will move to the first third after the first sorting. Thus it will not be involved in the second sorting. In the third sorting it will remain in the first third where it belongs.
- L **in thrid third** If L beings in the third third it will not be involved in the second sorting. After the second sorting L will move to the middle third. After the third sorting L will move to the first third where it belongs.
- E **in first third** If E begins in the first third of the list it will move to the second third after the sorting is over. In the second sorting E will move to the third third where it belongs. Thus it will not be involved in the third sorting.
- E **in second third** If E begins in the second third of the list it will remain in the second third after the first sorting. After the second screening E will move to the third third of the list where it belongs. Thus it will not be involved in the third sorting.
- E **in thrid third** If E beings in the third third of the list thus it will not be involved in the first sorting. After the second sorting it will remain in the third third where it belongs. Thus it will not be involved in the third sorting.
- M **in first third** If M begins in the first third of the list, after the first sorting it could remain in the first third or move to the second third. Either way after M will still be included in the third sorting because if M remains in the first third after the first sorting it will not be included in the sorting and if M moves to the second third it will remain there in the second sorting because only Es will in the third third as shown in previous cases. Since M is included in the third sorting and only Ls will be in the first third as shown in previous cases, then M will end in the second third where it belongs.
- M **in second third** If M begins in the second third of the list, after the first sorting it could move in the first third or remain to the second third. Either way after M will still be included in the third sorting because if M moves in the first third after the first sorting it will not be included in the sorting and if M remains to the second third it will remain there in the second sorting because only Es will in the third third as shown in previous cases. Since M is included in the third sorting and only Ls will be in the first third as shown in previous cases, then M will end in the second third where it belongs.
- M **in thrid third** If M beings in the third third of the list if will not be included in the first sorting. In the second sorting it will move to the second third because only Es will be in the third third after the second sorting as shown in the cases above. After the third sorting it will remain in the second third because only Ls will be in the first thrid as shown in the cases above. Thus M will end in the second third where it belongs

Since every number correctly arrives at the appropriate third of the list, assuming that the recursive sort works correctly, the numbres will be sorted in the correct order. Therefore StoogeSort correctly sorts the list.

Part 2: The running time of sorting 2/3 of the list is $T(n) = T(2n/3)$ because a recursive function is used. Thus the running time of StoogeSort overall is $T(n) = 3T(2n/3)$ where c represents adding some time constant to the fun time. Using master theorem on this recurrance sows that the asymptotic running time of StoogeSort is $O(n^{log_{\frac{3}{2}}3})$.

2) Solution

- $T(n) = T(n-1) + 3n - 3$
  By systematically bashing the recurrence equation I found to be exactly $\frac{3}{2}n^2 - \frac{3}{2}n + 1$. This can be proven with induction
  Base Case: $T(1) = 1$, $T(2) = T(1) + 6 - 3 = 1 + 6 - 3 = 4 = 6 - 3 + 1 = \frac{3}{2}2^2 - \frac{3}{2}2 + 1$
  Inductive Step: Assume for the sake of induction that $T(n) = T(n-1) + 3n - 3$ solves exactly to $\frac{3}{2}n^2 - \frac{3}{2}n + 1$ for n. $T(n+1) = T(n) + 3(n+1) - 3 = \frac{3}{2}n^2 - \frac{3}{2}n + 1 + 3n + 3 - 3 = \frac{3}{2}n^2 + 3n + \frac{2}{3} - \frac{3}{2}n - \frac{2}{3} + 1 = \frac{3}{2}(n+1)^2 - \frac{3}{2}(n+1) + 1$. Therefore by induction $T(n) = T(n-1) + 3n - 3$ solves exactly to $\frac{3}{2}n^2 - \frac{3}{2}n + 1$.

- $T(n) = 2T(n-1) + 2n - 1$
  By systematically bashing the recurrence equation I found to be exactly $3 * 2^n - 2n - 3$
  Base Case: $T(1) = 1$, $T(2) = 2T(1) + 2 * 2 - 1 = 2 + 4 - 1 = 5 = 12 - 4 - 3 = 3 * 2^2 - 2 * 2 - 3$
  Inductive Step: Assume for the sake of induction that $T(n) = 2T(n-1) + 2n - 1$ solves exactly to $3 * 2^n - 2n - 3$ for n. $T(n+1) = 2T(n) + 2(n+1) - 1 = 2(3 * 2^n - 2n - 3) + 2n + 2 - 1 = 3 * 2^{n+1} - 4n - 6 + 2n + 1 = 3 * 2^{n+1} - 2(n+1) - 3$. Therefore by induction $T(n) = 2T(n-1) + 2n - 1$ solves exactly to $3 * 2^n - 2n - 3$

- $T(n) = 4T(n/2) + n^3$
  Using Master Theorem shows that this is $O(n^3)$

- $T(n) = 17T(n/4) + n^2$
  Using Master Theorem shows that this is $O(n^{log_4 17})$

- $T(n) = 9T(n/3) + n^2$
  Using Master Theorem shows that this is $O(n^2 logn)$

- $T(n) = T(\sqrt{n}) + 1$
  Let there exist some S, k such that $S(k) = T(e^k)$. Thus $e^k = \sqrt{n}$ simplifes to $k = \frac{1}{2}logn$. Consider $S(k) = S(\frac{k}{2}) + 1$ which has asymptotic bound $O(logk)$. Therefore by subsitution $O(log(logn))$

3) Solution

Part 1:

Traverse through the k lists taking the first element from each list and adding it to a min heap. These k values will be the minimum of each list because the lists are sorted. Tag each vertex in the min heap with the number of the ith list where it was taken from. This step will take $O(k)$ time. Because the heap is a min heap, it will take $O(1)$ time to find the minimun vertex of the heap because it will be the left most vertex. Delete this vertex from the heap and add it to the beginning of an array making that vertex sorted. Reference the tag of this vertex to take the next number from the indicated array (the original ith where the number was taken from) and add it to the min heap. The min heap must then rearrange which

takes $O(logk)$ time. Repeat the process of removing the minimum vertex and inserting the next element from the ith list. This must happen n times making the final runtime $O(nlogk)$
To prove the correctness of this algorithm it is sufficent to show that the heap always contains the least element that has yet to be sorted because the heap will create the list by sequentally taking the minimum value of the heap. Initially it is easy to see that the heap contains the minimum value because it intializes with the minimum value of each of the list. When a number is removed the heap it is replaced with the next value from the ith list where removed number came from. This mimics how the heap was initialized because the heap now contains the minimum value of k lists, but now the total number of elements is lower. Therefore the heap always contains the minimum element of all the elements yet to be sorted.

Part 2:
Add the first k elements to a min heap. This step will take $O(k)$ time. Because the heap is a min heap, it will take $O(1)$ time to find the minimun vertex of the heap because it will be the left most vertex. Delete this vertex from the heap and add it to the beginning of an array. Take the next number from the unsorted list and add it to the min heap. The min heap must then rearrange which takes $O(logk)$ time. Repeat the process of removing the minimum vertex and inserting the next element from the unsorted list.This must happen n times making the final runtime $O(nlogk)$
To prove the correctness of this algorithm it is sufficent to show that the heap always contains the least element that has yet to be sorted because the heap will create the list by sequentally taking the minimum value of the heap. When intialized the heap must contain the minimum value of all the elements because the list is k-close meaning the first element must be within the first k elements of the unsorted list. Then when deciding the second lowest element of the list we add the $k+1$ element to the min heap. Allowing the second element to be selected from the $k+1$ first elements on the list, without the overall lowest element, guarenteeing the the second minimum element is in min heap. This argument can be applied over and over again to show that the ith lowest element of the list is choosen from the $k+i$ first elements of the list with the $i-1$ lowest elements already taken out, guarenteeing that the ith lowest element is in min heap. Therefore this algorithm works correctly.

4) Solution
This algorithm is a modifed form of the shortest paths on DAG's alogorithm presented in class (Lecture 4 Notes). Similarly start by using a DFS to topologically sort the DAG. The modification is before traversing across the edges to find the shortest path, traverse through all the edges and multiple the edge waits by -1. This makes the originally large weights small and the originally small weights large. Continue with the algorithm given in class to traverse through the edges to determine the shortest path of the modifed DAG with new edge saits. The single-source shortest path found of the modified DAG is actually the single-source longest path of the DAG. Since we must go through all the edges an extra time to multiply the weights by one the run time is modified from the original algorithm to be $O(m+|E|)$

5) Solution
Let the streets be represented by an undirected graph. This algorithm is a modification of DFS. The path can be created by the order of the *preorder* numbers and the *postorder* numbers with *preorder* representing walking one way down the street and *postorder* represnting walking the other way up the street. The only modification that needs to be applied is that when DFS encounters an edge to a vertex that has already been visited (a cycle), it will not traverse the edge towrds that vertex. In our algorithm, when a cycle occurs we will walk towards the already visited vertex and walk straight back, allowing us to be end in the same vertex and continue DFS as usual.

This algorithm will traverse all the edges in both directions because every vertex must have both a *preorder* number and a *postorder* number in DFS. This will take care of all edges that are do not create cycles. The edges that create cycles will be traverse in both edges by the modification made to DFS.

6) Solution

The main modification of Djikstra's is chanping the way we store information. Instead of using a priority heap this algorithm can implement Djikstra's with an array of sice $|V|m$. Information will be stored on the heap as follows. As verticies are traversed a label for that vertex will be placed at the index that is the current distance of the path. For example v_0 might be store at index 0 and if path to v_1 was of length 4, v_1 would be stored at index 4 and if v_2 was an additional 2 units away from v_1, v_2 would be stored in index 6. Because this is being placed in a specific index it takes time $O(1)$ to place vertices into the array. This algorithm will mimics the *deletemin* function in the Djikstra's pseudocode by searching the next m elements of the array after the vertex in question. If the vertex in question leads to another vertex, the next vertex would have to be located within the next m indices of the matrix because m is the maximum edge cost and there is no negative edge costs. Thus *deletemin* will have a run time of $O(m)$. Thus it takes $O(|V|m)$ to go through all the verticies of the graph. Therefore this algorithm has a $O(|E| + |V|m)$ run time overall.

7) Solution

By systematically bashing the recurrence equation I found to be exactly $n\lceil logn\rceil - 2^{\lceil logn\rceil} + 1$ where all *log*s are base 2. This algorithm can be proved with this induction below:
Base Case: $T(3) = T(\lceil 3/2\rceil) + T(\lfloor 3/2\rfloor) + 3 - 1 = T(2) + T(1) + 2 = 1 + 0 + 2 = 3 = 6 - 4 + 1 = 3\lceil log3\rceil - 2^{\lceil log3\rceil} + 1$
$T(4) = T(\lceil 4/2\rceil) + T(\lfloor 4/2\rfloor) + 4 - 1 = T(2) + T(2) + 3 = 1 + 1 + 3 = 5 = 6 - 4 + 1 = 3\lceil log4\rceil - 2^{\lceil log4\rceil} + 1$
Inductive Step:

- **Case 1: Even:** For easier notatation, assume for the sake of contradiction that the exact solution holds for $n-1$ where $n-1$ is odd. $T(n) = T(\lceil n/2\rceil) + T(\lfloor n/2\rfloor) + n - 1 = 2T(n/2) + n - 1 = 2(n/2\lceil logn/2\rceil - 2^{\lceil logn/2\rceil} + 1 = 2(n/2(\lceil logn/2\rceil + 1)) - 2^{\lceil logn/2\rceil+1} + 1 = 2 * n/2(\lceil logn/2\rceil + 1) - 2^{\lceil logn\rceil} + n + 1 = n\lceil logn\rceil - 2^{\lceil logn\rceil} + 1$. Therefore this exact solution holds for even numbers.

- **Case 2: Odd, 2^n+1:** Assume for the sake of contradition that the exact solution holds for n where n is a power of 2. $T(n+1) = T(\lceil (n+1)/2\rceil) + T(\lfloor (n+1)/2\rfloor) + (n+1) - 1 = \frac{n}{2}\lceil log\frac{n}{2}\rceil - 2^{\lceil log\frac{n}{2}\rceil} + 1 + (\frac{n}{2}+1)\lceil log(\frac{n}{2}+1)\rceil - 2^{\lceil log\frac{n}{2}\rceil+1} + 1 + n = \frac{n}{2}\lceil logn\rceil + (\frac{n}{2}+1)\lceil logn\rceil - 2^{\lceil logn\rceil-1} - 2^{\lceil logn\rceil} + n + 2 = n\lceil logn\rceil - \frac{n}{2} + \lceil logn\rceil + n + 1 - 2^{\lceil logn\rceil-1} - 2^{\lceil logn\rceil} + 1 = (n+1)\lceil log(n+1)\rceil - \frac{n}{2} - 2^{\lceil logn\rceil-1} - 2^{\lceil logn\rceil} + 1 = (n+1)\lceil log(n+1)\rceil - 2^{\lceil log(n+1)\rceil} + 1$ Therefore the exact solution holds for odd numbers which are in the form of $2^i + 1$.

- **Case 3: Odd, Not power of 2:** Assume fore the sake of contradition that the exact scolution holds for n where n is even but n is not a power of 2. $T(n+1) = T(\lceil (n+1)/2\rceil) + T(\lfloor (n+1)/2\rfloor) + (n+1) - 1 = T(n/2) + T(n/2+1) + n = \frac{n}{2}\lceil log\frac{n}{2}\rceil - 2^{\lceil log\frac{n}{2}\rceil} + 1 + (\frac{n}{2}+1)\lceil log\frac{n}{2} + 1\rceil - 2^{\lceil log\frac{n}{2}+1\rceil} + 1 + n = \frac{n}{2}\lceil log\frac{n}{2} - 1\rceil + (\frac{n}{2}+1)\lceil log\frac{n}{2} - 1\rceil - 2^{\lceil logn\rceil-1} - 2^{\lceil logn\rceil-1} + n + 2 = n\lceil logn\rceil - n + \lceil logn\rceil - 1 - 2^{\lceil log(n+1)\rceil} + 1 = (n+1)\lceil log(n+1)\rceil - 2^{\lceil log(n+1)\rceil} + 1$ Therefore the exact solutino holds for odd numbers which are not in the form of $2^i + 1$.

Therefore the exact colution holds for all n.