For all homework problems where you are asked to give an algorithm, you must prove the correctness of your algorithm and establish the best upper bound that you can give for the running time. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail. As always, try to make your answers as clear and concise as possible.

1. Buffy and Willow are facing an evil demon named Stooge, living inside Willow's computer. In an effort to slow the Scooby Gang's computing power to a crawl, the demon has replaced Willow's hand-designed superfast sorting routine with the following recursive sorting algorithm, known as StoogeSort. For simplicity, we think of Stoogesort as running on a list of distinct numbers. StoogeSort runs in three phases. In the first phase, the first 2/3 of the list is (recursively) sorted. In the second phase, the final 2/3 of the list is (recursively) sorted. Finally, in the third phase, the first 2/3 of the list is (recursively) sorted again.
Willow notices some sluggishness in her system, but doesn't notice any errors from the sorting routine. This is because StoogeSort correctly sorts. For the first part of your problem, prove rigorously that StoogeSort correctly sorts. (Note: in your proof you should also explain clearly and carefully what the algorithm should do and why it works even if the number of items to be sorted is not divisible by 3. You may assume all numbers to be sorted are distinct.) But StoogeSort can be slow. Derive a recurrence describing its running time, and use the recurrence to bound the asymptotic running time of Stoogesort.

A. We will use proof by strong induction to prove the correctness of Stoogesort.

**Before we prove our base cases**, let us review terminology of Stoogesort. We will review other principles of Stoogesort after the base cases have been proven.

*The 3 Phases and 3 Thirds of Stoogesort:* In the Stoogesort method, there are 3 phases, each of which calls Stoogesort on a list that is of length $\frac{2}{3}$ the size of the original list. Each phase excludes a particular third of the original list. Thus, we can divide the total list into thirds and refer to each third differently. We can refer to the lowest third of the list as the <u>Left Third</u>, the middle third of the list as the <u>Middle Third</u>, and the highest third of the list as the <u>Right Third</u>. Phase 1 and Phase 3 calls Stoogesort on the Left and Middle third. Phase 2 calls Stoogesort on the Middle and Right third.

**Let us prove our base cases:**
$b_0$*:* We have a list of length one. A list of length one is trivially sorted.
$b_1$ : We have a list of length two.
If the list is already sorted, then Phases 1, 2, and 3 of the stoogesort will not swap the order of the list. If the list is not sorted, then when Phase 1 will grab 2/3 of 2. When the algorithm grabs 2/3 of a number not divisible by 3, it will always round up to the nearest integer. Continuing in phase 1, it thus will grab both elements and swap them. Phases 2 and 3 will not swap the two elements again, and the list will be sorted.
$b_2$*:* We have a list of length 3. We can refer to the values of the list as L-low, M-medium, and H-high.
If the order is LMH, Stoogesort will not swap the order.
If the order is MLH, Stoogesort will sort the list to LMH in Phase 1 and will not swap the order after.
If the order is LHM, Stoogesort will sort the list to LMH in Phase 2 and will not swap the order in Phases 1 or 3.
If the order is HLM, Stoogesort will sort the list to LHM in Phase 1. It will sort the list to LMH in Phase 2. It will not swap the order in Phase 3.
If the order is MHL, Stoogesort will not sort the list in Phase 1. It will sort the list to MLH in Phase 2 and LMH in Phase 3.
If the order is HML, Stoogesort will sort the list to MHL in Phase 1, MLH in Phase 2, and LMH in Phase 3.

In all permutations of a list of length 3, the list will be correctly sorted. Thus base case 2 yields a correct sort.

**Inductive Hypothesis:** Stoogesort sorts for all lists of sizes 1 - (n - 1).
**Inductive step:** We will prove that Stoogesort sorts for all lists of sizes n.

*"Belonging" to a third:* If we have a list of size n and that list has been correctly sorted, when a particular element j is correctly sorted, we can say that j "belongs" to the third that it is in. Ex: j belongs to the Left Third.

*The Significance of Correct Position:* $\frac{2}{3}n$ is less than n - 1 for all n > 2. Our base cases include n=1 and n=2. Thus, we know that when Stoogesort is called on a list that is sized $\frac{2}{3}n$, that list must be sorted correctly based upon the inductive hypothesis. However, before Stoogesort is called recursively on the list $\frac{2}{3}n$ in size, we need to guarantee that all elements belonging to each third of the list sized n has been positioned into the correct third and will not be moved after it has been positioned. If we can prove that any arbitrary element belonging to any third has been correctly positioned and maintained in the same third, then due to the inductive principle used upon recursion, Stoogesort will correctly sort the list size n.

Our inductive step will prove exactly that: any arbitrary element $i$ belonging to any third will be correctly positioned and maintained after the 3 phases of Stoogesort are run on a list size n.

We have three cases for which examining an arbitrary element $i$:
**Case I:** $i$ belongs to the Left Third.
**Case II:** $i$ belongs to the Middle Third.
**Case III:** $i$ belongs to the Right Third.

We analyze our subcases depending on $i$'s initial starting position and the phase of the algorithm:
**Each cases's Phase 1 starting and resulting positions:**
**Case I:**
**Subcase A:** $i$ starts in the Left or Middle Third $\rightarrow$ $i$ is in the correct position (of the Left and Middle Thirds) when Stoogesort is called on the list $\frac{2}{3}n$. *Thus, i will be correctly positioned into the Left Third and i will not be removed from the Left Third*
**Subcase B:** $i$ starts in the Right Third $\rightarrow$ $i$ will not be touched after Phase 1.
**Case II:**
**Subcase A:** $i$ starts in the Left or Middle third $\rightarrow$ $i$ will end Phase 1 in the position of Left or Middle third. $i$'s correct positioned is not guaranteed yet.
**Subcase B:** $i$ starts in the Right Third $\rightarrow$ $i$ will not be touched after Phase 1.
**Case III:**
**Subcase A:** $i$ starts in the Left or Middle third $\rightarrow$ $i$ will end Phase 1 in the position of the Middle Third. There are only a limited number of elements greater than $i$, so $i$ must be sorted to the far right of its sorted list, which in the case corresponds to the Middle.
**Subcase B:** $i$ starts in the Right Third $\rightarrow$ $i$ will not be touched after Phase 1.

**Each cases's Phase 2 starting and resulting positions:**
**Case I:**
**Subcase A':** i is already correctly positioned: $i$ has been positioned into the correct third and will not be moved to another third.
**Subcase B':** $i$ will be in the Right Third$\rightarrow$ $i$ will be sorted into the Middle Third after Phase 2.
**Case II:**
**Subcase A':** $i$ will be in the Left Third$\rightarrow$ $i$ will not be touched after Phase 2
**Subcase B':** $i$ will be in the Middle or Right Third$\rightarrow$ $i$ will be correctly positioned in the Middle Third.

Because Phase 2 solidifies the Right Third (Phase 2 Case III below supports this), Subcase B' elements, though they may be moved within the Middle Third during Phase 3, are guaranteed position and guaranteed to not be moved out of the Middle Third.
**Case III:**
**Subcase A'∪B':** $i$ will start in the Middle or Right third$\rightarrow$ This case is symmetric to **Case I Subcase A:** Because $i$ is in the correct position (of the Middle and Right Thirds) when Stoogesort is called on the list $\frac{2}{3}n$ of Phase 2, *$i$ will be correctly positioned in the Right Third.*

**Each cases's Phase 3 starting and resulting positions:**
**Case I:**
**Subcase B":** $i$ will be in the Middle Third$\rightarrow$ $i$ will be positioned into the Left Third. This case is equivalent to Case I A and shares the same reasoning that *$i$ will be positioned correctly.*
**Case II:**
**Subcase A∪B":** $i$ will start in the Middle Third$\rightarrow$ Because Phase 2 solidified the highest numbers in the Right Third and Phase 3 solidifies the lowest numbers in the Left Third, Phase 3 also solidifies the Middle Third to its correct positioning as well. Thus, *$i$ will be correctly positioned.*

All cases and subcases support that for a list of size n, after one run of Stoogesort, any arbitrary element, regardless of which third it belongs to, will be positioned into the correct Third of the list before Stoogesort is called on $\frac{2}{3}n$, which will correctly sort that list. Thus, our strong inductive step holds and Stoogesort is proven to be correct.

Recurrence Relation: Stoogesort is run 3 different times on a list of size n and the recurrence is run on a $\frac{2}{3}n$. When merging a list of size n, that takes linear time:

$$T(n) = T(\frac{2}{3}n) + T(\frac{2}{3}n) + T(\frac{2}{3}n) + T(merge(n))$$

$$= 3 * T(\frac{2}{3}n) + cn$$

Plugging this into the master theorem, we require a and b to be whole numbers. Because, as n approaches infinite, the difference between $\frac{2}{3}n$ and $\frac{1}{2}n$ is unimportant, we can round our b value, which would be $\frac{2}{3}n$, to 2.

Thus, a = 3, b = 2 and, k = 1.
Because $a > b^k$, T(n) = O($n^{\log_2 3}$)

2. (Part A) Solve the following recurrences exactly, and then prove your solutions are correct by induction. (Hint: Graph values and guess the form of a solution: then prove that your guess is correct.)

• $T(1) = 1, T(n) = T(n-1) + 4n - 4$

Solution:

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

*Proof.* Let us use proof by induction. Base Case: We show that the solution holds for $b_0$ where n = 1.

$$b_0 : T(1) = 1 : T(1) = 2 * 1 - 2 * 1 + 1 = 1$$

Inductive Hypothesis: For all natural numbers that are less than or equal to k, the solution holds: $T(k) = 2k^2 - 2k + 1$ Inductive Step: We are trying to prove that:

$$T(k+1) = 2(k+1)^2 - 2(k+1) + 1$$

We know that:

$$T(k+1) = T(k) + 4(k+1) - 4 = T(k) + 4k$$

We can plug in our solution for T(k):

$$T(k+1) = 2k^2 - 2k + 1 + 4k$$

$$= 2k^2 + 4k + 2 - 1 - 2k - 1 + 1$$

$$= 2(k^2 + 2k + 1) - 2k - 2 + 1$$

$$= 2(k+1)^2 - 2(k+1) + 1$$

We have shown that this solution holds for T(k + 1). Because our solution holds for the base case and the inductive step, by mathematical induction, the statement T(n) holds for all natural numbers n. ■

• $T(1) = 1, T(n) = 2T(n-1) + 2n - 1$

Solution:

$$T(n) = 6 * 2^{n-1} - 2n - 3$$

*Proof.* Let us use proof by induction to prove that Stoogesort sorts correctly. Base Case: We show that the solution holds for $b_0$ where n = 1.

$$b_0 : T(1) = 1 : T(1) = 6 * 1 - 2 * 1 - 3 = 1$$

Inductive Hypothesis: For all natural numbers that are less than or equal to k, the solution holds: $T(k) = 6 * 2^{k-1} - 2k - 3$ Inductive Step: We are trying to prove that:

$$T(k+1) = 6 * 2^k - 2(k+1) - 3$$

We know that:

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

We can plug in our solution for T(k):

$$T(k+1) = 2 * (6 * 2^{k-1} - 2k - 3) + 2k + 1$$

$$= 6 * 2^k - 4k - 6 + 2k + 1$$

$$= 6 * 2^k - 2k - 2 - 3$$

$$= 6 * 2^k - 2(k+1) - 3$$

We have shown that this solution holds for T(k + 1). Because our solution holds for the base case and the inductive step, by mathematical induction, the statement T(n) holds for all natural numbers n. ■

(Part B) Give asymptotic bounds for T(n) in each of the following recurrences. Hint: You may have to change variables somehow in the last one.

- $T(n) = T(n/2) + n^3$

$$a = 4, b = 2, k = 34 < 2^3, T(n) = O(n^3)$$

- $T(n) = 17T(n/4) + n^2$

$$a = 17, b = 4, k = 217 > 4^2, T(n) = O(n^{\log_4 17})$$

- $T(n) = 9T(n/3) + n^2$

$$a = 9, b = 3, k = 2, 9 = 3^2, T(n) = O(n^2 logn)$$

- $T(n) = T(\sqrt{n}) + 1$.

$$T(n) = T(n^1/2) + 1); n = 2^k, k = log_2 n$$
$$T(2^k) = T(2^{\frac{k}{2}}) + 1$$
$$T(2^k) = T'(k)$$
$$T'(k) = T(\frac{k}{2}) + 1, a = 1, b = 2, k = 0, 1 = 2^0$$
$$T'(k) = O(logk)$$
$$T(2^k) = T'(k) = O(log(log_2 n))$$
$$T(n) = O(log(logn))$$

3. Design an efficient algorithm to find the longest path in a directed acyclic graph. (Partial credit will be given for a solution where each edge has weight 1; full credit for solutions that handle general real-valued weights on the edges, including negative values.)

For this problem, we will begin with our weighted DAG G with vertices and edges (V, E). We will modify the Algorithm Shortest Paths 2 Function and the Procedure Update Function from lecture notes. We will turn the Algorithm Shortest Paths 2 Function into the Longest Path Function and turn the Procedure Update Function into the Procedure Big Update Function.

1. We will change the (Single Source) Algorithm Shortest Paths 2 function into an Longest Path Function by renaming the dist array the max and initially setting max[V] values for every vertex in the graph equal to $-\infty$. We will change the Procedure Update Function's conditional line from $dist[v] > dist[w] + length(w, v)$ to $dist[v] < dist[w] + edgeWeight(w, v)$ to make it the Procedure Big Update Function.
2. The algorithm begins by topologically sorting the DAG G and saving that into an array.
3. Then for every vertex, if the vertex has no incoming edges (basically if the vertex is a source), we will run the Longest Path Function with our Procedure Big Update Function. The algorithm will save the largest max[V] value for that vertex in a separate array called ultimateMax that is the size of the number of vertices in the graph.
4. The algorithm will compare all ultimeMax values and it will return the largest value followed by its corresponding vertex.

**Runtime:** Topological sort has an upper bound runtime of O($|V| + |E|$). The Longest Paths function has a runtime of O($|V| + |E|$) (as proved in lecture). Because the worst case scenario involves each vertex being a source, the upper bound is O($V(|V| + |E|) + |V| + |E|$) or O($V^2 + |V| * |E|$).

*Proof.* To prove the correctness of this algorithm, we use proof by induction. We are running the Single Source *Longest* Path Algorithm on every source of the graph. In lecture, we have proved the correctness of a topologial sort. This proof needs to prove the conditional statement of the Procedure Big Update function per source to guarantee that every time a maximum value (maximum weighted distance value) is updated, it really is the maximum value of that graph. We will induct upon the number of edges we have evaluated of the graph.

**Base case 0:** We have not evaluated the edges of any vertex or we have evaluated 0 edges of the graph. By not evaluating any edges, we have not traversed any edges, so the max value of our starting vertex is 0 and the max value of every other vertex is $-\infty$.

**Base case 1:** If the first edge of the first vertex exists, then the evaluation and update based upon edge 1 will be correct: The max value of the connecting vertex is $-\infty$. Thus, the algorithm must update the max value of the connecting vertex to be equal to the value of the edge's weight plus 0. Assuming no edge value can be equal to $-\infty$, the max value of the connecting vertex will be updated correctly.

**Inductive Hypothesis:** If we check and evaluate exactly k edges in which k ¡ n, then all k of those edges will correctly update the max values stored for each corresponding vertex.

**Inductive Step:** We will evaluate edge n of arbitrary vertexes (U, V)and prove that the nth edge will correctly update the max value for V. We can assume that prior to nth edge's approaching update, all max values of the graph for this specific source have been updated correctly. We also know that the edge weight of the nth edge (U, V) will be correctly represented. Thus, when the max value of the V is compared to the max value of the U plus the weight value of n ((U, V)), we know that all variables in our updating equation are correct. Thus, we know the max value of the V will be correctly updated according to the nth edge.

Because every update of the max weight algorithm will be correct for every edge, the function will yield a correct longest path algorithm for each source. When comparing all source's longest paths, the algorithm will correctly output the longest path of the DAG. ■

4. Giles has asked Buffy to optimize her procedure for nighttime patrol of Sunnydale. (This takes place sometime in Season 2.) Giles points out that proper slaying technique would allow Buffy to traverse all of the streets of Sunnydale in such a way that she would walk on each side of each street, exactly once, going up the street in one direction and down the street in the other direction. Buffy now has slayer homework: how can it be done? (If you have to assume anything about the layout of the city of Sunnydale, make it clear!)

We will assume that Sunnydale can be represented as a directed graph. Each street corner is represented by a vertex, and each street of Sunnydale is represented by two edges–one going in one direction, the other in the opposite. Buffy can only travel streets going in the "forwards" direction, for safety purposes. Every street corner has at least two edges and street corners cannot have an odd number of edges. The graph of Sunnydale is strongly connected: it wouldn't make sense if there was some street corner that you couldn't get to from another street corner–humans drive cars, not planes. Buffy needs to cross each street on both sides in opposite directions, essentially traversing all streets of Sunnydale. She will do so using a Depth First Search.

1. Buffy has a little notebook. In that notebook, Buffy is going to record that she has not visited any of the street corners. She will also create a Buffy Stack.
2. Because any street corner is accessible from any other street corner, Buffy is going to choose whatever street corner she likes best to start with. Let's call this Street Corner V. Buffy is going to record that she has visited V, and she is going to put V on her Buffy Stack.
3. For each street that V is connected to, Buffy is going to perform the following tasks: She will pick one street (arbitrarily) and cross the street in the forwards direction.
4. When she gets to the other side of the street, she sees she is at street corner W.
   (a) Looking at her records, if Buffy has visited W, then she is going to turn around, and return to V on the other side of the street, going again in the forwards direction (safety).
   (b) If Buffy hasn't visited W, then she is going to put W on her Buffy Stack. She is going to repeat steps 3-5 for Street Corner W.
5. If Buffy has visited all of V's streets, she is going to cross V off her list and remove V from her Buffy Stack. Then, she is going to look at the name of the street corner that is now at the top of her Buffy Stack, and travel to that street corner (going on in forwards direction of the road; she's a safe lady).

The fact that we are representing the city as a DAG should not affect or prevent Buffy from traversing every street. She will treat all edges as potential paths and travel only in the forwards direction. Additionally, Buffy won't need to "fly" anywhere because she of the strongly directed connection of the streets.

There are two different ways that Buffy handles streets. Those can be distinguished whether one side of that street is a tree edge of the DFS or whether it's not. In a DFS, every edge is either a tree edge or it is not. In Buffy's Sunnydale, every street either has one tree-edge or it does not.
When Buffy encounter a street with a tree-edge (U,V), we know that she has added U to her Buffy Stack and travelled forwards along (U, V). When Buffy take her street edge off her Buffy Stack, in this case U, she traverses the street in the "forwards" direction. The "forwards" direction this time is the opposite direction from which she initially came, so that would be Buffy travelling across edge (V, U). Thus, if Buffy encounters a street with a tree-edge, she is guaranteed to walk along both sides of that street (go forwards on both of the street's edges).

Buffy also walks both ways on streets that do not have a single tree edge: when Buffy checks if she has already explored a street corner (let's say she is at Vertex X checking Vertex Y), she always traverses forwards one way (X, Y) and forwards the other way as well (Y, X).

Thus, because Buffy goes forwards in both directions on every street with one tree edge and every street without a tree edge, Buffy has gone in both directions on every street exactly once and has secured a very safe Sunnydale.

Just as was the case with the regular DFS, the upper bound of Buffy's (DFS) algorithm is O($|V| + |E|$).

5. Explain how to solve the following two problems using heaps. (No credit if you're not using heaps!) First, give an O(nlogk) algorithm to merge k sorted lists with n total elements into one sorted list. Second, say that a list of numbers is k-close to sorted if each number in the list is less than k positions from its actual place in the sorted order. (Hence, a list that is 1-close to sorted is actually sorted.) Give an O(nlogk) algorithm for sorting a list of n numbers that is k-close to sorted.

A. The algorithm is written in steps below. Because of the algorithm's repetition, the steps are numbered. We assume that the initially sorted lists are distinct. We assign an index to each initially sorted list and can refer to each by that index of 0 – (k-1).

1. Create a min heap list of size k.
2. Create a final sorted list of size n.
3. From each of the k initially sorted lists, pop the first element and add each to the min heap list.
4. Run Min-Heapify(min heap list, min heap list[1]) followed by Extract-Min(min heap list) to sort and then extract the smallest value of the min heap list. Take that min value, from initially sorted list list j, and add it to the final sorted list.
5. If the jth initially sorted list is not empty, pop its first element and add it to the min heap list.
6. If the min heap list is not empty, repeat step steps 4-6, else return the final sorted list.

Looking at this algorithm for time complexity, we can record the time for each step:
1, 2) O(1) to create lists and variables
3) O(k) to add k elements to a list.
4) Min-Heapify has a runtime of log(length of heap). We are adding 1 element from k lists to create a heap list of length k. Running Min-Heapify will thus take O(logk). Extracting min will take O(1). This step will take O(logk) + O(1) which ultimately equals O(logk).
5) Inserting an element will take O(1).
6) We will add all of our total elements to our min heap one by one eventually, and thus will run min-heapify followed by extract min a total of n times to create a list of length n.

Adding this up and multiplying 5-6 by n per 7's repetition request, our final runtime is $2 * O(1) + O(k) + n(O(1) + O(logk)) = O(nlogk + n + k + c)$. Because nlogk has the highest degree or is the dominating term of this equation, we can ignore all terms other than nlogk when looking for an upper bound.
**Thus, our algorithm is O(nlogk).**

**At the end of the algorithm**, we will receive a final sorted list called list S. We will prove S's correctness:

*Proof.* Looking at two arbitrary elements, S[i] and S[i + 1], we must prove that S[i] is less than S[i + 1]. There are two total cases for S[i] to precede S[i + 1] in the sorted list S.
**Case I:** S[i] and S[i + 1] belonged to the same initially sorted list.
**Case II:** S[i] and S[i + 1] belonged to different initially sorted lists.

**Case I:** S[i] and S[i + 1] belonged to the same initially sorted list.
If S[i] and S[i + 1] are members of the same initial sorted list, then S[i + 1] was added to the min heap after S[i] was extracted from the min heap. S[i + 1] was added to the min heap because it proceeded S[i] in the ordering of the initially sorted list. Due to the definition of a sorted list, S[i] must have a smaller value than S[i + 1].

**Case II:** S[i] and S[i + 1] belonged to different initially sorted lists.
When S[i] was placed into the finally sorted list, it is because the functions Min-Heapify and Extract Min yielded S[i] as the smallest value of a min heap, lets call it min heap B, at that point in the algorithm.

Because S[i + 1] follows S[i] in the final sorted listed and S[i + 1] was not apart of the same initially sorted list as S[i], then S[i + 1] must have been a member of B. Because B, after being sorted by Min-Heapify, yielded S[i] as its minimum rather than S[i + 1], by definition of a min heap, S[i] must be less than S[i + 1].

Because we have proved that S[i] is less than S[i + 1] for any two arbitrary elements in our final sorted list, we have proved the correctness of our sorting algorithm. ■

B. Second, say that a list of numbers is k-close to sorted if each number in the list is less than k positions from its actual place in the sorted order. (Hence, a list that is 1-close to sorted is actually sorted.) Give an O(nlogk) algorithm for sorting a list of n numbers that is k-close to sorted.
The algorithm is written in steps below. Because of the algorithm's repetition, the steps are numbered.

1. Create a final sorted list of size n.
2. Create a min heap list of size k.
3. Add the (0 - (k-1)) elements of the k-close to sorted list to the min heap list.
4. Create a counter element i and set i equal to k.
5. Call Min-Heapify(min heap list, min heap list[1]) and call Extract Min. Add the min value to the end of the final sorted list.
6. If i is less than n, add the ith element of the close to sorted list to the min heap. Increment i.
7. Repeat steps 5-6 until the min heap list is empty.

*Proof.* This algorithm functions by sorting the array smallest to largest. The algorithm grabs a section that is sized k and finds the minimum element of that section using a min heap. The algorithm then stores that element in the ith index in the final sorted array. That element is the ith smallest number in the array (this statement entertains a zero index). We will prove that every time the algorithm grabs/heapifies a section for the ith index of the final array, the ith smallest element is guaranteed to be in that section. If the smallest element is always guaranteed to be sorted first into the final array and the array is sequentially sorted smallest to largest, then the sorting algorithm will be correct. We will induct upon the index of the element added to the final array.

**Base case: i = 0.** We begin with the 0th smallest element that belongs to the 0th index of the final sorted array. This smallest element is at most k - 1 positions away from the position 0. The algorithm grabs a section of the array from 0 to k - 1. Because 0 + k - 1 = k - 1, the smallest element must be in the range of 0 to k - 1. Thus, the 0th smallest element is sorted in the min heap and placed into index 0.

**Base case: i = 1.** The algorithm is now looking for an element for the 1st index of the final sorted array. The algorithm grabs the kth element and adds it to the heap. Because the 0th element was already removed from the heap, the minimum of this new heap, the heap that now also involves the kth element, is guaranteed to be the next smallest element of the final sorted list, which in this case is 1.

**Inductive Hypothesis:** Every element less than n is sorted into the correct index of the final array. The min heap used to extract element n is called B.

**Inductive Step:** It is known that all previous $(0 - n)$ indexes in the final sorted list are correct. It is also known that know that the previous minimum that was just extracted was correctly the nth index of the final sorted list. Based upon those two facts, the farthest left element of the k-close to sorted list that could contain n + 1 that did not contain a preceding minimum must be in the heap. Then, the n + kth (the $n + 1 + k - 1$th) element is added to the minimum heap. Now, the farthest right element of the k-close to sorted list that could contain the n + 1th minimum has been added to the heap. Because all of the prior minimums are out of the heap and all possible n + 1th minimums are in the heap, the heap's newest minimum must be the n + 1th element.

Because the inductive step proves that the minimum element will always be contained in the heap and correctly sorted into the final sorted array, the correctness of this sorting algorithm has been proven. ■