Sorted Runtimes

Here is a video walkthrough of the solutions.

We want to sort an array of N **unique** numbers in ascending order. Determine the best case and worst case runtimes of the following sorts:

(a) Once the runs in merge sort are of *size* $<= N/100$, we perform insertion sort on them.

Best Case: $\Theta(\qquad)$, Worst Case: $\Theta(\qquad)$

Solution:
Best Case: $\Theta(N)$, Worst Case: $\Theta(N^2)$

Once we have 100 runs of size N/100, insertion sort will take best case $\Theta(N)$ and worst case $\Theta(N^2)$ time. The constant number of linear time merging operations don't add to the runtime.

(b) We can only swap adjacent elements in selection sort.

Best Case: $\Theta(\qquad)$, Worst Case: $\Theta(\qquad)$

Solution:
Best Case: $\Theta(N^2)$, Worst Case: $\Theta(N^2)$

The best case and worst case don't change since swapping at most doubles the work each iteration, which produces the same asymptotic runtime as normal selection sort.

(c) We use a linear time median finding algorithm to select the pivot in quicksort.

Best Case: $\Theta(\qquad)$, Worst Case: $\Theta(\qquad)$

Solution:
Best Case: $\Theta(N\log(N))$, Worst Case: $\Theta(N\log(N))$

Doing an extra N work each iteration of quicksort doesn't asymptotically change the best case runtime, but it improves the worst case runtime to be the same as the best case. Recall the best case runtime of quicksort is $\Theta(N\log(N))$. You may wonder, why don't we always do this then? Well, there are a couple reasons. First, if the initial array is randomly sorted, the worst case behavior is very improbably. Second, the added linear work per level doesn't change anything asymptotically, but it does slow down the algorithm in practice.

(d) We implement heapsort with a min-heap instead of a max-heap. You may modify heapsort but must maintain constant space complexity.

Best Case: $\Theta(\qquad)$, Worst Case: $\Theta(\qquad)$

Solution:
Best Case: $\Theta(N\log(N))$, Worst Case: $\Theta(N\log(N))$

While a max-heap is better, we can make do with a min-heap by placing the smallest element at the right end of the list until the list is sorted in

descending order. Once the list is in descending order, it can be sorted in ascending order with a simple linear time pass.

(e) We run an optimal sorting algorithm of our choosing knowing:

- There are at most N inversions

  Best Case: $\Theta($ $\quad$ $)$, Worst Case: $\Theta($ $\quad$ $)$

  Solution: Best Case: $\Theta(N)$, Worst Case: $\Theta(N)$

  Recall that insertion sort takes $\Theta(N + K)$ time, where K is the number of inversions. If K is at most N, then, insertion sort has the best and worst case runtime of $\Theta(N)$. Here is an explanation for why no sorting algorithm can surpass this. Notice for our algorithm to terminate we *either* need to address every inversion or look at every element. Since there are at most N inversions, knowing that we have addressed every inversion would take us at least $\Theta(N)$ time. Looking at every element in the list would also take us $\Theta(N)$ time. In either case, we see the runtime of any sorting algorithm cannot be faster than $\Theta(N)$.

- There is exactly 1 inversion

  Best Case: $\Theta($ $\quad$ $)$, Worst Case: $\Theta($ $\quad$ $)$

  Solution: Best Case: $\Theta(1)$, Worst Case: $\Theta(N)$

  The inversion may be the first two elements, in which case constant time is needed. Or, it may involve elements at the end, in which case N time is needed. It can be proven quite simply that no sorting algorithm can achieve a better runtime than above for the best and worst case.

- There are exactly $(N^2 - N)/2$ inversions

  Best Case: $\Theta($ $\quad$ $)$, Worst Case: $\Theta($ $\quad$ $)$

  Solution: Best Case: $\Theta(N)$, Worst Case: $\Theta(N)$

  If a list has $N(N - 1)/2$ inversions, it means it is sorted in descending order! So, it can be sorted in ascending order with a simple linear time pass. We know that reversing any array is a linear time operation, so the optimal runtime of any sorting algorithm is $\Theta(N)$.