(a) We run merge sort, but once the inputs for the recursive calls are of *size* $<= N/100$, we perform insertion sort on them.

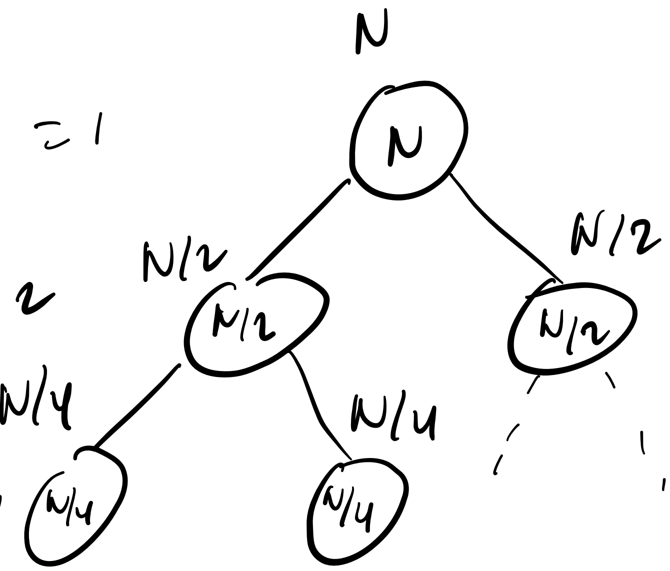
Best Case: $\Theta($ $\quad\quad\quad$ $)$, Worst Case: $\Theta($ $\quad\quad\quad$ $)$

mergeSort(list) {
left = mergeSort(left half of list)
right = mergeSort(right half of list)
merge(left, right) takes $\Theta(N)$ time
}

$2^0 = 1$

$2^1 = 2$

$2^2 = 4$



log N levels, each level does N work

We run merge sort, but once the inputs for the recursive calls are of *size* $<= N/100$, we perform insertion sort on them.

2^3 $N/8$

2^4 $N/16$

2^5 $N/32$

2^6 $N/64$

2^7 $N/128$ → how many nodes at this level?

$2^\ell = 2^7$ sublists

Takes $8N \Rightarrow \Theta(N)$ time to get to this!

We know that we perform Insertion Sort on each of these 2^7 sublists

$\therefore$ Runtime

Best case $= 2^7 \cdot$ Best case Insertion

$= 2^7 \cdot \Theta(N) = \Theta(N)$

Worst case $= 2^7 \cdot$ Worst case Insertion

$= 2^7 \cdot \Theta(N^2) = \Theta(N^2)$

$\therefore$ Since it takes $8N \Rightarrow \Theta(N)$ time for Merge Sort to get sublists of size $<= N/100$, we add this in our total runtime:

	Merge		Insertion		
Best Case :	$\Theta(N)$	+	$\Theta(N)$	=	$\Theta(N)$
Worst Case :	$\Theta(N)$	+	$\Theta(N^2)$	=	$\Theta(N^2)$

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 run selection sort, but instead of swapping the smallest element to the front, we can only swap adjacent elements.

Best Case: Θ( ), Worst Case: Θ( )

```
SelectionSort(list) {
    for each element in list:   (N time)
        find min element in rest of list   (roughly N time)
        swap positions of current elem and min elem   (constant)
}
```

Runtime: $N + N-1 + N-2 + \ldots + 1 = \Theta(N^2)$

Constraint: Can only swap adjacent elements now, rather than doing one swap b/w curr elem and min elem

i.e. 1 2 7 4 0 3

This means swapping is no longer a constant time operation in selection sort!

```
SelectionSort(list) {
    for each element in list:   (N time)
        find min element in rest of list   (roughly N time)
        swap positions of current elem and min elem   (now N time)
}
```

Before : iterating $\cdot$ (findMin + swap curr and min)

$N \cdot (N + 1) \Rightarrow \Theta(N^2)$

After : iterating $\cdot$ (findMin + adjacent swap min to front)

$N \cdot (N + N) \Rightarrow N \cdot 2N \Rightarrow 2N^2 = \Theta(N^2)$

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

Quicksort(list) {

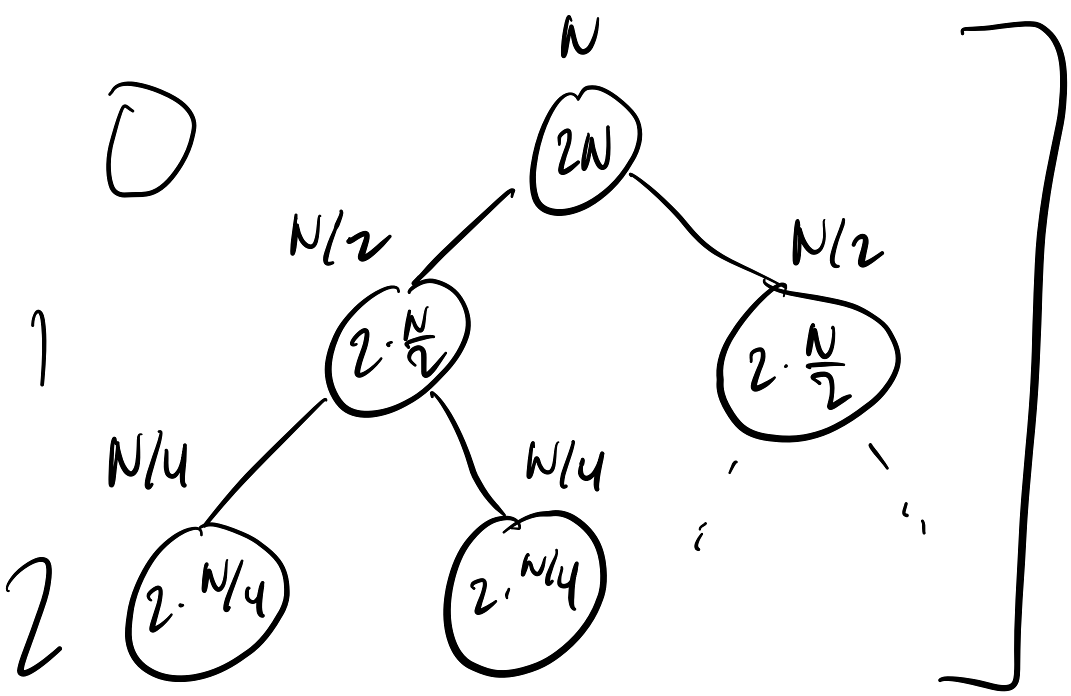
choose pivot (median algo takes N time)

partition elems about pivot (N time)

quicksort(less) + pivot + quicksort(greater)

}

Since pivot is median value, left and right partitions will be split evenly, i.e. both will be of size $\simeq N/2$



log N levels

$2^{l} = 2^{2} = 4$

Input split in half due to median pivot

$$\text{Sum of work} = 2N + 2\cdot\left(2\cdot\frac{N}{2}\right) + 4\cdot\left(2\cdot\frac{N}{4}\right) + \ldots$$

$$= 2N + 2N + 2N + \ldots + 2N$$

each level does 2N work, log N levels

$$\therefore\ 2N\cdot\log N = \Theta(N \log N)$$

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.

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

Best Case: Θ(   ), Worst Case: Θ(   )

The items are unique!

```
HeapSort(list) {
    heapify list into Max heap (takes N time)
    while heap is not empty: (takes N time)
        Swap root w/ last elem
        bubbleDown new root ] (takes log N time)
}
```

However, by using a min-heap, the resulting list will be in descending order, since we add the minimum to the end of the list

We have to then reverse the list to get it in ascending order.

```
minHeapSort(list) {
    heapify list into min-heap (takes N time)
    while heap is not empty: (takes N time)
        Swap root w/ last elem
        bubbleDown new root ] (takes log N time)
    reverse list (takes N time)
}
```

$\therefore$ $N + N\log N + N = \boxed{\Theta(N \log N)}$

Since all items are unique, no diff b/w best and worst case since bubble downs are $\log N$ time.

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 : pair of elems (X, Y) X comes before Y and $X > Y$

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

Insertion Sort : $\Theta(N + K)$, $K = \#$ inversions

Worst Case : $\leq N$ inversions

$$\Theta(N + N) \Rightarrow \Theta(N)$$

Best Case : 0 inversions

$$\Theta(N) \Rightarrow \Theta(N)$$

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

Best case! inversion pair is first 2 elems in list

5	2				

Takes constant time to find and swap! $\Rightarrow \Theta(1)$

Worst Case: inversion pair is last 2 elements in list

				5	2

Takes N time to find, constant time to swap! $\Rightarrow \Theta(N)$

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 $\frac{N(N-1)}{2}$ inversions

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

Consider an array of size $N = 4$

$$N = 4 \Rightarrow \frac{4(4-1)}{2} = 6 \text{ inversions}$$

How to create 6 inversions in this array?

- Let's consider finding max # of inversions

A	B	C	D

Pretend A, B, C, D are arbitrary values

What is the max # of inversions for

A : 3

B : 2

C : 1

D : 0

	A	B	C	D	
max inversions:	3	2	1	0	= 6 inversions!
	$N-1$	$N-2$	$N-3$	... 0	

$$\frac{N^2 - N}{2} = \frac{N(N-1)}{2} = \text{Sum of first } N\text{-}1 \text{ \#'s}$$

3	2	1	0

3 2 1 0 = 6 inversions!

To have $\frac{N(N-1)}{2}$ inversions means the array is in descending order!

Thus, the optimal algorithm is to reverse the list! $\Rightarrow$ $\Theta(N)$ time

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 $\Theta(N)$.