1\. First, set the left boundary, left, to be 1
set the right boundary, right, to be n
start search.

Search (left, right):
if left = right = 1
if $A[1] = 1$
return 1, the index is found
else
no such index, search ends

else if right - left = 1

if $A[left] = left$
return left, the index is found
else if $A[right] = right$
return right, the index is found
else
no such index, search ends

else find the integer with index $i = \frac{left + right}{2}$

if $A[i] > i$
set $right = i - 1 = \frac{left + right}{2} - 1$
start next search (left, $\frac{left + right}{2} - 1$)
else if $A[i] < i$
set $left = i + 1 = \frac{left + right}{2} + 1$
start next search ($\frac{left + right}{2} + 1$, right)
else $A[i] = i$
return $i = \frac{left + right}{2}$, the index is found

In the worst case, we would keep searching in the array until we reach a search range of size 2, which shows right - left = 1, and give the answer, either get the value of index or the index does not exist.

Since we divide the search range by 2 at each iteration:

$$T(n) \longrightarrow T(\frac{n}{2})$$

It is a binary search and the total times of iteration is $\sim \log_2 n$, so its complexity would be $O(\log n)$, which is the running time in the worst case.

2.(a) $T(n) = 8T(\frac{n}{2}) + cn^4$

node size	# of nodes	work/node	total work
$S = n$	1	$c \cdot s^4$	$c \cdot n^4$
$S = \frac{n}{2}$	8	$c \cdot s^4$	$8 \cdot c \cdot (\frac{n}{2})^4$
$S = \frac{n}{4} = \frac{n}{2^2}$	$64 = 8^2$	$c \cdot s^4$	$8^2 \cdot c \cdot (\frac{n}{2^2})^4$
...	...	...	...

total work (assume $n = 2^k$)

$$T(n) = cn^4 (1 + 8 \cdot (\tfrac{1}{2})^4 + 8^2 \cdot (\tfrac{1}{2^2})^4 + \cdots + 8^k \cdot (\tfrac{1}{2^k})^4)$$

$$= cn^4 (1 + \tfrac{1}{2} + \tfrac{1}{2^2} + \cdots + \tfrac{1}{2^k})$$

$$= cn^4 \frac{1 - \frac{1}{2}(\frac{1}{2})^k}{1 - \frac{1}{2}}$$

$$= 2cn^4 (1 - \tfrac{1}{2}(\tfrac{1}{2})^{\log_2 n})$$

$$= 2cn^4 (1 - \tfrac{1}{2n})$$

$$= 2cn^4 - cn^3$$

$$\in O(n^4)$$

(b) $T(n) = 4T(\frac{n}{2}) + dn^4$

node size	# of nodes	work/node	total work
$s = n$	1	$d \cdot s^4$	$d \cdot n^4$
$s = \frac{n}{2}$	4	$d \cdot s^4$	$4 \cdot d \cdot (\frac{n}{2})^4$
$s = \frac{n}{4} = \frac{n}{2^2}$	$16 = 4^2$	$d \cdot s^4$	$4^2 \cdot d \cdot (\frac{n}{2^2})^4$
...	...	...	...

total work (assume $n = 2^k$)

$$T(n) = dn^4(1 + 4 \cdot (\frac{1}{2})^4 + 4^2 \cdot (\frac{1}{2^2})^4 + \cdots + 4^k \cdot (\frac{1}{2^k})^4)$$

$$= dn^4(1 + \frac{1}{4} + \frac{1}{4^2} + \cdots + \frac{1}{4^k})$$

$$= dn^4 \frac{1 - \frac{1}{4}(\frac{1}{4})^k}{1 - \frac{1}{4}}$$

$$= \frac{4}{3}dn^4(1 - \frac{1}{4}(\frac{1}{4})^{\log_2 n})$$

$$= \frac{4}{3}dn^4(1 - \frac{1}{4n^2})$$

$$= \frac{4}{3}dn^4 - \frac{1}{3}dn^2$$

$$\in O(n^4)$$

(c) The second algorithm is not asymptotically better than the first one, since their big-O asymptotic runtime are both $O(n^4)$.

I think there are two reasons for this outcome :

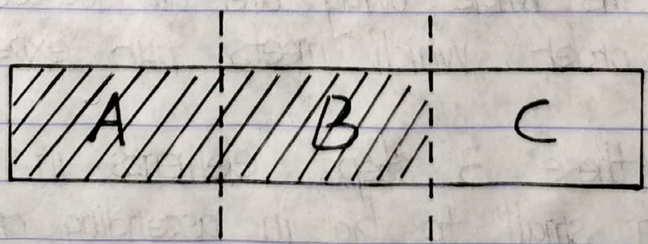
1. both of the algorithms solve a problem of size n by dividing it into half size $\frac{n}{2}$

$$T(n) \longrightarrow T(\frac{n}{2})$$

2. both of the algorithms combine the solutions of subproblems in time $O(n^4)$

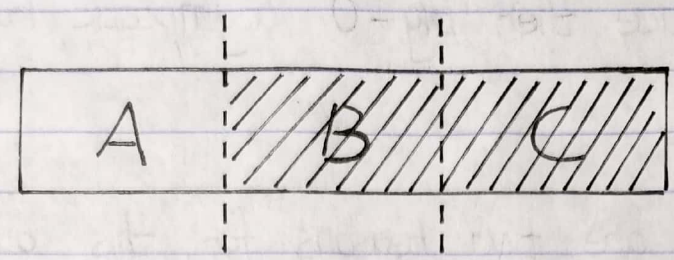
$$cn^4 \ \& \ dn^4$$

3.(a) 1° sort the first two-thirds of the array



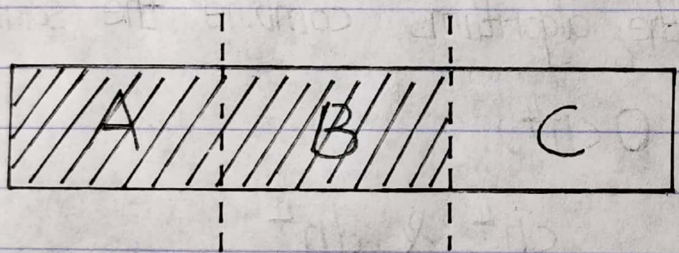
after sorting, bigger elements in the first two-thirds of the array are all located in region B in ascending order.

2° sort the last two-thirds of the array



after sorting, bigger elements in the whole array are all located in region C in ascending order, which meets our expectation.

3° sort the first two-thirds of the array again



after sorting, smaller elements in the whole array are all located in region A in ascending order, and mediate elements in the whole array are all located in region B in ascending order, which meets our expectation.

After all these 3 steps, elements in the array are sorted from small to big in ascending order.

Therefore, Stooge Sort successfully yields a sorted array.

(b) Stooge Sort solves a problem of size n by dividing it into 3 subproblems of size $\frac{2}{3}n$, recursively solving each subproblem, then combining the solutions in constant time $O(1)$.

recurrence relation : $T(n) = 3T(\frac{2}{3}n) + c$ (c is constant)

(c)

node size	# of nodes	work/node	total work
$S=n$	1	c	c
$S=\frac{2}{3}n$	3	c	$3\cdot c$
$S=\frac{4}{9}n=(\frac{2}{3})^2n$	$9=3^2$	c	$3^2\cdot c$

total work (assume $n=(\frac{3}{2})^k$)

$$T(n) = c\cdot(1+3+3^2+\cdots+3^k)$$
$$= c\cdot\frac{1-3\cdot3^k}{1-3}$$
$$= \frac{c}{2}(3\cdot3^k-1)$$
$$= \frac{c}{2}(3\cdot3^{\log_{\frac{3}{2}}n}-1)$$
$$= \frac{c}{2}(3\cdot n^{\log_{\frac{3}{2}}3}-1)$$
$$= \frac{3}{2}c\cdot n^{\log_{\frac{3}{2}}3}-\frac{c}{2}$$
$$\in O(n^{\log_{\frac{3}{2}}3}) \sim O(n^{2.7})$$

(d) The worst-case runtime :

merge sort : $O(n \log n)$

selection sort : $O(n^2)$

insertion sort : $O(n^2)$

bubble sort : $O(n^2)$

quick sort : $O(n^2)$

Stooge sort : $O(n^{2.7})$

Stooge Sort is the slowest algorithm.