HW 4

Derrick DeBose
ID# 1536268

8.1-1) If the comparison sort is already in sorted order then $n-1$ comparisons will be made and that means the smallest possible depth of a leaf in a decision tree is $\boxed{n-1}$.

8.1-2) $\sum_{k=1}^{n} \lg k$ $\qquad$ $\lg(n!) = \Theta(n \lg n)$

Proof: BigO $\quad \sum_{k=1}^{n} \lg k \leq \sum_{k=1}^{n} \lg n = n \lg n = O(n \lg n)$

Proof: BigOmega $\quad \sum_{k=1}^{n} \lg k = \sum_{k=1}^{\lfloor n/2 \rfloor} \lg k + \sum_{k=\lfloor n/2 \rfloor + 1}^{n} \lg k$

$$\geq \sum_{k=n/2}^{n} \lg k$$

$$\geq \sum_{k=n/2}^{n} \lg n/2$$

$$\geq (n/2) \lg(n/2)$$

$$= \Omega(n \lg n)$$

$$\therefore \boxed{\sum_{k=1}^{n} \lg k = \Theta(n \lg n)}$$

8.2-1)

A = | 6 | 0 | 2 | 0 | 1 | 3 | 4 | 6 | 1 | 3 | 2 |

	0	1	2	3	4	5	6
C =	2	2	2	2	1	0	2

	0	1	2	3	4	5	6
C =	2	4	6	8	9	9	11

	1	2	3	4	5	6	7	8	9	10	11
B =						2					

	1	2	3	4	5	6	7	8	9	10	11
B =			1		2		3				

	1	2	3	4	5	6	7	8	9	10	11
B =				1		2	3	3	4		6

	1	2	3	4	5	6	7	8	9	10	11
B =		0	1	1	2	2	3	3	4		6

	0	1	2	3	4	5	6
C =	2	4	5	8	9	9	11

	0	1	2	3	4	5	6
C =	2	3	5	7	9	9	11

	0	1	2	3	4	5	6
C =	2	3	5	6	8	9	10

	0	1	2	3	4	5	6
C =	1	2	4	6	8	9	10

SORTED! B = | 0 | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 6 | 6 |

	0	1	2	3	4	5	6
C =	0	2	4	6	8	9	9

8.2-2) Since the counting sort starts at the back of the array lines (10-12) when its inputing the storage of the A-array into the B-array, the C-array holds where each number should be placed in the B-array. Since the B-array is insorted order as they are seen at the end of the array A the output array is in the same order as they do in the input array. So counting sort is stable.

8.3-1) RADIX SORT

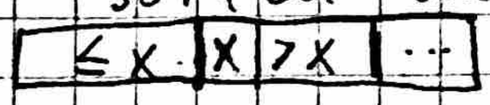
COW	SEA	THB	BAR
DOG	TEA	BAR	BIG
SEA	MOB	EAR	BOX
RUG	TAB	TAR	COW
ROW	DOG	SEA	DIG
MOB	RUG	TEA	DOG
BOX →	DIG →	DIG →	EAR
TAB	BIG	BIG	FOX
BAR	BAR	MOB	MOB
EAR	EAR	DOG	NOW
TAR	TAR	COW	ROW
DIG	COW	ROW	RUG
BIG	ROW	NOW	SEA
TEA	NOW	BOX	TAB
NOW	BOX	FOX	TAR
FOX	FOX	RUG	TEA

SORTED!

6.3-2)

	Stable?	
insertion sort	yes	
mergesort	yes	
heap sort	no	(order is forgotten in binary tree)
quicksort	no	(swaps non-adjacent elements)

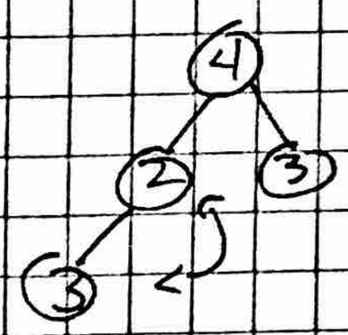
the insertion step in insertion sort



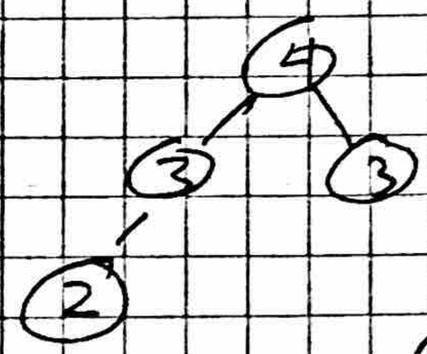
when x is placed into the sorted part of the array it stops in the location where all values are above x in the sorted portion and the ≤ x ensures that if x is equal to the lower bound it won't go past that equal value. If it were < x rather than ≤ x then insertion sort wouldn't be stable.

Merge sort is stable when implemented correctly. The Merge step is when data is being moved around in the sort. If the left and right arrays being compared have the same value, then in order for merge sort to be stable it must take the data from the left array before it takes it from the right array.

Heapsort is not a stable sort. There is the list $(4, 2, 3_1, 3_2)$



when maxheapify gets called the 2 and 3 switch



Max heapify switches the last 3 and 2 and that breaks stability $(4, 3_2, 3_1, 2)$

Quicksort is not stable.

Consider list $(4_1, 4_2, 2)$

during the partition step

4_1 and 2 will get swapped

causing for the order to be

$(2, 4_2, 4_1)$ breaking stability.

8.3-3) Radix-Sort(A,d)

for i=1 to d
use a stable sort to sort array A on digit i

Intialization: The for loop starts with the least significant digit and uses a stable sort to sort that array on that digit i.

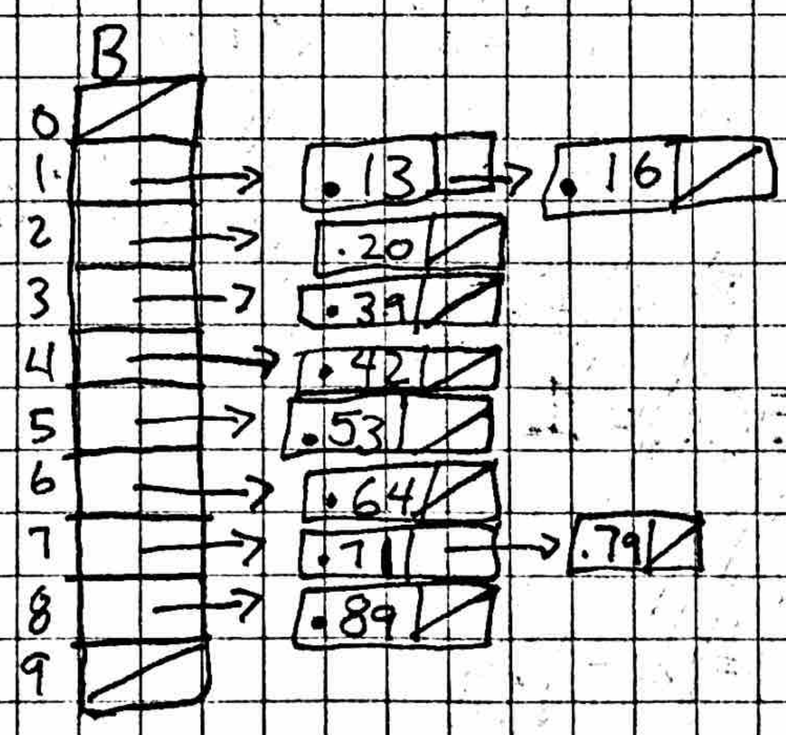
Maintenance: As the loop sorts based on the ones place and then sorts based on the tens place. If our sort method is not stable then the data won't be properly in the right order during each interation through the loop. By ensuring the sort is stable we know we properly sort the ones place then the tens place all the way up until the d^{th}-digit.

Termination: The loop terminates when i=d+1. So, we know that the invarient holds and the numbers are sorted based on d-digits.

8.4-1) A=(.79,.13,.16,.64,.39,.20,.89,.53,.71,.42)

A

1	.79
2	.13
3	.16
4	.64
5	.39
6	.20
7	.89
8	.53
9	.71
10	.42



8.4-2) The worst case running time for bucket sort is $\Theta(n^2)$.

The case this happens is when the input goes into only one bucket and the data is in reverse order. This is because insertion sort runs in $\Theta(n^2)$ time.

Since we know our input data can have any formation of numbers between $[0, 1)$ then if we use merge sort instead of insertion sort in line 8 then the worst case run time would be $O(n \lg n)$.

8.4-3) $P(X=0) = 1/4$ TT

$P(X=1) = 1/2$ HT, TH

$P(X=2) = 1/4$ HH

$E[X] = 0 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} = \frac{1}{2} + \frac{1}{2} = 1$

$E[X^2] = 0^2 \cdot \frac{1}{4} + 1^2 \cdot \frac{1}{2} + 2^2 \cdot \frac{1}{4} = \frac{1}{2} + 2 = \boxed{\frac{3}{2}}$

$E^2[X] = 1^2 = \boxed{1}$

8-4) a) deterministic algorithim would be just to compare each red pair with each blue pair.

This results in $\Theta(n^2)$ comparisons

b) If we put each set of jugs into a decision tree then there are n-permutations that map red jugs to blue jugs. for the length of the tree l we get the inequality

$$n! \leq l \leq 2^h$$

$$\Omega(n \lg n) \leq \lg(n!) \leq h$$

c) We will use a similar approach to randomized quicksort

1) randomly choose a red pivot $O(1)$

2) find the matching blue pivot $O(n)$

3) Partition the blue jug around the red pivot and put the blue pivot in place $O(n)$

4) Partition the red jug around the blue pivot and put the red pivot in place $O(n)$

The expected time of this algorithm is $O(n \lg n)$ because the partition is expected to happen somewhere near the middle of the array and that means there will be $O(n \lg n)$ comparisons.

The worst number of comparisons is $\Theta(n^2)$ because we could get a bad partition that reduces the size to $n-1$ each iteration.