8.1.1 ***What is the smallest possible depth of a leaf in a decision tree for a comparison sort?***
for n elements we have $n-1$ pair of relative ordering, what leads us to $n-1$ as the smallest possible depth

---

8.1.2 ***Obtain asymptotically tight bound on $lg(n!)$ without using Stirling's approximation. instead, evaluate the summation $\sum_{k=1}^{n} lg\ k$***

We can use integrals to solve it, since $\lg$ *its a monotonically incrasing function we have*:

$$\int_0^n \lg(x) \le \sum_{k=1}^{n} lg\ k \le \int_1^{n+1} \lg(x)\,dx \to \frac{nln(n)-n}{\ln(2)} \le \sum_{k=1}^{n} lg\ k \le \frac{(n+1)\ln(n+1)-n}{\ln(2)}$$
$$\le \frac{(n+1)\ln(n+1)}{\ln(2)} \to$$

$$\frac{nln(n)-n}{\ln(2)} \le \sum_{k=1}^{n} lg\ k \le \frac{(n+1)\ln(n+1)}{\ln(2)} \to n \cdot \frac{\lg_2 n}{\ln(2)\lg_2 e} - \frac{n}{\ln(2)} \le \sum_{k=1}^{n} lg\ k$$
$$\le \frac{(n+1)\lg_2(n+1)}{\lg_2 e \ln(2)} \to$$

$$n \cdot lg(n) - \frac{n}{\ln(2)} \le \sum_{k=1}^{n} lg\ k \le (n+1)\lg_2(n+1)$$

now we need to prove the following: $c_1 nlg(n) \le nlg(n) - \frac{n}{\ln(2)} \to$

$$nlg(n)[1-c_1] \ge \frac{n}{\ln(2)}.\ if\ n>0 \to \lg(n)\,[1-c_1] \ge \frac{1}{\ln(2)} \to \lg(n) \ge \frac{1}{\ln(2)\,[1-c_1]}$$

We know that $\lg(1)=0$ *and* $\lg(2)=1$, *so for* $n \ge 4$ *we need to find a* c_1 *such that*

$2\ln(2)\,[1-c_1] \ge 1 \to 2-2c_1 \ge \frac{1}{\ln(2)} \to c_1 \le 1 - \frac{1}{2\ln(2)}$, *we can find such* c_1

so we discover that for $c_1 \le 1 - \frac{1}{2\ln(2)}$ *and* $n \ge 4$ *we have*:

$$c_1 nlgn \le n \cdot lg(n) - \frac{n}{\ln(2)} \le \sum_{k=1}^{n} lg\ k \to c_1 nlgn \le \sum_{k=1}^{n} lg\ k$$

Now we need to show that: $(n+1)\lg_2(n+1) \le c_2 nlg(n) \to$ *lets analyze the limit of*:

$\lim_{n\to\infty} \frac{(n+1)\lg(n+1)}{nlog(n)} = 1$. *Based on that there exists such* c_2 *for some* $n_0 \to$

$$c_1 nlgn \le \sum_{k=1}^{n} lg\ k \le c_2 nlg(n)\ \forall\ n \ge \max\{4, n_0\}$$

8.1.3 ***Show that there is no comparison sort whose running time is linear for atleast half of the n! inputs of length n. What about a fraction of $\frac{1}{n}$ of the inputs of length n? What about a fraction of $\frac{1}{2^n}$***

If we run linear for $\frac{n!}{2}$ *cases then we would have* $\frac{n!}{2}$ *leaves in a linear height*

In this situation we have a height h*, that have* 2^h *leaves at most, so* $\frac{n!}{2} \leq 2^h$

$h \geq \lg\left(\frac{n!}{2}\right) = \lg(n!) - 1 = \Theta\big(nlg(n)\big)$*what is a contraction*

The same goes for the other cases:

$h \geq \lg(n!) - \lg(n) = \Theta\big(nlg(n)\big) - \Theta(\lg(n)) = \Theta(nlgn)$

$h \geq \lg\left(\frac{n!}{2^n}\right) \rightarrow \lg(n!) - n = \Theta\big(nlg(n)\big) - \Theta(n) = \Theta\big(nlg(n)\big)$

8.1.4 ***You are given an n element input sequence, and you know in advance that it is partlysorted in the following sense. Each element initially in position i such that i mod*** $\mathbf{4} = \mathbf{0}$ ***is either already in its correct position, or it is one place away from its correct position. For example, you know that after sorting, the element initially in the position*** $\mathbf{12}$ ***belongs in position*** $\mathbf{11}, \mathbf{12}$ ***or*** $\mathbf{13}$***. You have no advance information about the other elements, in position i where i mod*** $\mathbf{4}! = \mathbf{0}$***. Show that an*** $\boldsymbol{\Omega(nlgn)}$ ***lower bound on comparison − based sorting still holds in this case***

If we proceed as usual clearly the property will hold, now if we use this advantage

We know that in n elements we will have $\left\lfloor\frac{n}{4}\right\rfloor$ *such special positions.*

possibly occupying $3\left\lfloor\frac{n}{4}\right\rfloor$ *or* $\left\lfloor\frac{n}{4}\right\rfloor + 2$*. For these itens we actually onyl need to realize at most two comparisons, given that the rest is already sorted*

so we have $n - \left\lfloor\frac{n}{4}\right\rfloor$ *to perform a sort what counts to* $\lg\left(\left(n - \left\lfloor\frac{n}{4}\right\rfloor\right)!\right)$ *comparisons*

plus $2 * \left\lfloor\frac{n}{4}\right\rfloor$ *comparisons. for the possible values of n we have*:

if n is divisible by $4 \rightarrow \lg\left(\frac{4n - n + 0}{4}!\right) = \lg\left(\frac{3n}{4}!\right)$

if $n + 1$ *is divisible by* $4 \rightarrow lg\left(\frac{4n - n + 3}{4}!\right) = \lg\left(\frac{3n + 3}{4}!\right)$

if $n + 2$ *is divisible by* $4 \rightarrow \lg\left(\frac{4n - n + 2}{4}!\right) = \lg\left(\frac{3n + 2}{4}!\right)$

if $n + 3$ *is divisible by* $4 \rightarrow \lg\left(\frac{4n - n + 1}{4}!\right) = \lg\left(\frac{3n + 1}{4}!\right)$

they are all $\Theta(nlg(n))$

8.2.1 ***Using figure*** $\mathbf{8.2}$ ***as a model, illustrate the operation of*** $\boldsymbol{COUNTING - SORT}$ ***on the array*** $\boldsymbol{A} = \{\mathbf{6}, \mathbf{0}, \mathbf{2}, \mathbf{0}, \mathbf{1}, \mathbf{3}, \mathbf{4}, \mathbf{6}, \mathbf{1}, \mathbf{3}, \mathbf{2}\}$

This will be left as a training exercise for the reader

8.2.2 *Prove that Counting Sort is stable*

```
COUNTING-SORT(A, n, k)
1  let B[1:n] and C[0:k] be new arrays
2  for i = 0 to k
3      C[i] = 0
4  for j = 1 to n
5      C[A[j]] = C[A[j]] + 1
6  // C[i] now contains the number of elements equal to i.
7  for i = 1 to k
8      C[i] = C[i] + C[i − 1]
9  // C[i] now contains the number of elements less than or equal to i.
10 // Copy A to B, starting from the end of A.
11 for j = n downto 1
12     B[C[A[j]]] = A[j]
13     C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14 return B
```

To prove it is stable it only has a meaning if the number is associated with some sateliteData
In this sense we will say that each element of A is associated with object A_i and they are all different until line 10. what we have done is to put in $C[i]$ for each i, the number of elements Smallers or equal than the value i
Now we assume that counting sort works, now lets analyze the loop in lines 11 to 13:
Let $A[c]$ and $A[k]$ be equal elements and without loss of generality that $c > k$
Since They are the same element $C[A[c]]$ access the same memory addres that $C[A[k]]$
But since $A[c]$ was acessed before $C[A[c]] > C[A[k]]$ since in the end of for loop
we decrement the value and never add it it again. Then when we assign the value $B\left[C[A[k]]\right]$
$C[A[k]]$ will be in a location lower, so if $k \leq c \rightarrow C[A[k]] \leq C[A[c]]$,
this proves that its a stable algorithm

8.2.3 *Suppose that we were to rewrite the for loop header in line 11 of the COUNTING − SORT as 11: for $j = 1$ to n*
Show that the algorithm still works properly, but that it is not stable. Then rewrite the pseudocode for countingsort so that elements with the same value are written into the output array in order of increasing index and the algorithm is stable

First assume that it works properly, in this situation lets make the same analysis as before:
let $A[c] = A[k]$ and let $c < k$ without loss of generality. In this situation
$C[A[c]]$ access the same memory as $C[A[k]]$ but since $c < k$, $C[A[c]]$ is $\geq$ than $C[A[k]]$.
This makes that if $c < k$ then $A[c] > A[k]$, translating it, two equal keys with associated data, that one comes before, like A_c comes before A_k will get a position in B such that A_k will be before A_c. It's therefore not stable

Now lets prove that the algorithm still holds: Since we are just reversing the equal values
The sorting is still valid

Since we just reversed the order of the equal elements we need to know how many elements are greater or equal In this situationwe can increment the memory address

$So\ let$:
$COUTING\ SORT(a, N, K)$:
$let\ B[1:n] and\ C[0:k] be\ new\ arrays$
$for\ i = 0\ to\ k$
  $C[i] = 0$
$for\ j = 1\ to\ n$
  $C[A[j]] = C[A[j]] + 1$
$for\ i = k\ to\ 1$:
  $C[i] = C[i] + C[i+1]$
$for\ j = 1\ to\ n$:
  $B\left[C[A[j]]\right] = A[j]$
  $C[A[j]] = C[A[j]] + 1$

8.2.4 ***Prove the following loop invariant for COUNTING − SORT*:**
***At the start of each iteration of the for loop of lines* 11 − 13 *the last element in A with value i that hast not yet copied into B belongs to B*[*C*[*i*]]**

Initialization: $At\ the\ first\ iteration\ we\ haven'tcopied\ anything\ yet\ and\ we\ are\ in\ the\ situation$
$In\ which\ the\ C\ array\ contains\ the\ number\ of\ elements\ that\ are\ lower\ than\ i\ for\ each$
$0 \le i \le k. In\ this\ case\ the\ last\ element\ in\ A\ with\ value\ i, that\ has\ not\ been\ copied$
$Will\ obviously\ go\ to\ B[C[i]]. Since\ its\ the\ first\ time\ we\ access\ the\ memory\ i\ of\ the\ C\ array$

Maintenance: $The\ maintenance\ holds\ with\ similarity. Suppose\ that\ this\ happens\ for\ 1, n - 1. Now\ lets\ prove\ it\ holds\ for\ n. For\ a\ given\ interaction, before\ the\ actual\ one, we\ have$
$A[j] = i\ and\ it\ goes\ to\ B[C[i]]\ place\ lets\ get\ the\ rightmost\ element\ before\ A[j]$
$with\ value\ i, if\ there\ is\ none, its\ okay, if\ there\ is\ atleast\ one\ we\ know\ that\ Since\ this\ item$
$is\ the\ next\ one\ and\ in\ the\ actual\ iteration\ we\ will\ decrement\ C[i] by\ one\ we\ are\ done.$

Termination: $It\ follows\ immediately\ for\ the\ maintenance$

8.2.5. ***Suppose that the array being sorted contains only integers in the range* 0 *to k and that there are no satellite data to move with those keys. Modify counting sort to use just arrays A and C. putting the sorted result back into array A instead of into a new array B***

$Since\ there\ is\ no\ satellite\ data\ involved\ there\ is\ no\ meaning\ to\ maintain\ it\ stable$
$In\ this\ situation\ we\ can\ only\ count\ up\ each\ element\ and\ put\ its\ number\ in\ the\ C\ array$
$After\ that\ we\ fill\ the\ A\ array\ with\ the\ number\ from\ C$:
$let\ C[0:k]\ be\ a\ \ new\ arrays$
$for\ i = 0\ to\ k$
  $C[i] = 0$
$for\ j = 1\ to\ n$
  $C[A[j]] = C[A[j]] + 1$
$c = 1$
$for\ j = 1\ to\ k$:
  $for\ i = 1\ to\ C[A[j]]$
    $A[c] = j; c++$

8.2.6 ***Describe an algorithm that, given, n integers in the range 0 to k, proprocesses its input and then answers any query about how many of the n integers fall into a range*** $[a:b]$ ***in*** $O(1)$ ***time. Your algorithm should use*** $\Theta(n+k)$ ***preprocessing***

$for\ i = 0\ to\ k$
  $C[i] = 0$
$for\ j = 1\ to\ n$
  $C[A[j]] = C[A[j]] + 1$
$for\ i = 1\ to\ k$:
  $C[i] = C[i] + C[i-1]$
$return\ C[b] - C[a-1]$

8.2.7 ***Counting sort can also work efficiently if the input values have fractional parts, but the number of digits in the fractional part is small. Suppose that you are given n nunbers in the range 0 to k, each with at most d decimal (base 10) digits to the right of the decimal point. Modify counting sort to run in*** $\Theta(n + 10^d k)$.

The idea here is to multiply *this values by* 10^d *in this manner we endup with only integers In this situation* $10^d k + 1$ *possible locations in* C, *so the location of an element* $A[j]$ *is not* $C[A[j]]$ *but* $C[A[j] \cdot 10^d]$

8.3.1 ***Using figure 8.3 as a model. Illustrate the operation of RadixSort on the following list of English Words COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX***

From the left to right

COW	SEA	TAB	BAR
DOG	TEA	BAR	BIG
SEA	MOB	EAR	BOX
RUG	TAB	TAR	COW
ROW	RUG	SEA	DIG
MOB	DOG	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

8.3.2 ***Which of the following sorting algorithms are stable*: *Insertion sort*, *mergesort*, *heapsort and quicksort*? *Give a simple scheme that makes any comparison sort stable*. *How much additional time and space does your scheme entail*?**

MergeSort can be stable by simpling choosing the element from the left array if the one from the left and the one from the right are equal

InsertionSort is stable because we compare elements from a highr index with subarrays with lower indices and only change if they are higher, for the moment they are equal we maintain
HeapSort is not stable because we don't have control over the other they appear, we are only based on certain comparisons
Quicksort may also be not stable depending on the partition method we use
One method is to store the index of each element in this situation when we compare them and they are equal we compare the index. In the worst case all elements are equal and use compare each pair
Two times more, because its a constat we doesn'tchange the complexity of the algorithm
The aditional space is n, we double it(in terms of units)

8.3.3 ***Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable*?**
We will use the inductive hypothesys that in the end of the loop the columns from i to 0 *are already sorted considering the subnumbers of the number* $[i:0]$. *With that*
In the end of the first iteration we will call a stable sorting algorithm that will sort these numbers so it's okay for the first case. Now lets suppose it holds for all until $n-1$. *This means that* $[n-1:0]$ *are sorted. Then we will call the sorting algorithm*
What leads us to two situation if no two elements are equal, then $[n:0]$ *will be obviously sorted, now, if two elements are equal and since one comes before the other, we know that without loss of generality , Because the elements are ordered by* $[n-1:0]$ *subnumbers, the one that comes before is such that* $A[k][n-1:0] \leq A[j][n-1:0]$
Since this is a stable sort algorithm we endup with $A[k][n] = A[j][n]$ *and since* $A[k][n-1:0] \leq A[j][n-1:0]$, $A[k]$ *will be before*$A[j]$ *because of the stability and based on that* $A[n:0]$*will still be sorted. We then need the stability to maintain the order given the sort on a digit d may result in such digits being equal*

8.3.4 ***Suppose that Counting − Sort is used as the stable sort within Radix −Sort. If Radix − Sort calls COUNTING − SORT d times, then since each call of COUNTING − SORT makes two passes over the data (lines 4 − 5 and 11 − 13), altogheter 2d passes over the data occur. Describe how to reduce the total number of passes to d + 1***

We still need atleast d passes through the data because for each digit − we need to pass through the elements to display them at the right place

The thing is we use the others passes just to put on the array C how many numbers are less or equal than a value, we still need to do this, but we can for all digits in just one loop
Obvisouly the number of operations will be larger, but we still can reduce it just one loop through the data we can for example, change the lines $4-5$ *adding a inner loop*:

```
for i = 1 to n:
    for digit = 1 to d:
      ...do stuff..
```

8.3.5 ***Show how to sort n integers in the range*** $0 \to n^3 - 1$ ***in*** $O(n)$ ***time***

Let the number $n^3 - 1$ *be* b *− bits long in this situation we know that b is equal to:* $\lceil \lg_2 n^3 - 1 \rceil$. *Now is* $\lceil \lg_2 n^3 - 1 \rceil \geq \lfloor \lg(n) \rfloor$ *because* $\lg_2 n^3 - 1 > \lg(n)$, *because* $2 \log_2 n - 1 > 0$. *In this situation we choose a subdivision of* $r = \lfloor \lg(n) \rfloor$ *is optimal*

Our running time becomes so $\Theta\left(\frac{\lceil \lg_2 n^3 - 1 \rceil}{\lfloor \lg(n) \rfloor}\left(n + 2^{\lfloor \lg(n) \rfloor}\right)\right) \sim \Theta(n)$

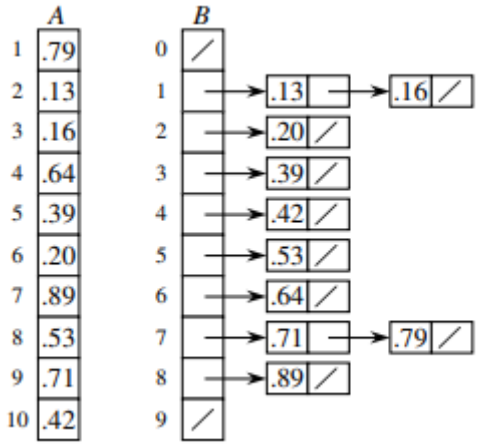
We can still do a better analysis, if we transform each element to base n we have 3 digits since its base n for each column we have n possible values, our counting sort becomes $O(2n)$ *We do 3 times counting sort what leads us to still* $O(n)$

8.3.6 ***In the first card − sorting algorith in this section, which sorts on the most significant digit first, exactly how many sorting passes are needed to sort*** d ***− digit decimal numbers in the worst case? How many piles of cards does an operator need to keep trcack of in the worst case?***

The thing with this is that even if a number is lower than the other for a given digit i We need to place it based on the higher digits too. So in this situation you have to keep track of Just one value that is incremeted every iteration in a loop

---

8.4.1 ***Using Figure*** **8.4** ***as a model, illustrate the operation of Bucket − Sort on the array*** $A = [.79, .13, .16, .64, .39, .20, .89, .53, .71, .42]$



8.4.2 ***Explain why the worst case running time for bucket sort is*** $\Theta(n^2)$. ***What simple change to the algorithm preserves its linear average − case running time and makes its worst − case running time*** $O(nlgn)$?

```
BUCKET-SORT(A, n)
1  let B[0 : n − 1] be a new array
2  for i = 0 to n − 1
3      make B[i] an empty list
4  for i = 1 to n
5      insert A[i] into list B[⌊n · A[i]⌋]
6  for i = 0 to n − 1
7      sort list B[i] with insertion sort
8  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
9  return the concatenated lists
```

By the code above we verify that we need to use $\Theta(n)$*work for all lines except possibly for the loop in line* 6*, because of insertion sort. Since all elements can go to the same bucket*
We can basically implement a insertion sort using BucketSort with no pros it all,
Since the worst case of Inserion Sort is $O(n^2)$ *the worst* $-$ *case running time is* $O(n^2)$
Our expected running time for this algorithm is as follow:
$E[T(n)]$
$= \Theta(n) + \sum_{i=0}^{n-1} E\left[O\left(f(n_i)\right)\right]$ *where* $f(n)$*represents the function of* n *for a choosen algorithm to perform such sort, we can use MergeSort ou HeapSort for example*
Since its average and worst case are all lower than insertion sort we preserve the characteristics

---

8.4.3 ***Let X be a random variable that is equal to the number of heads in two flips of a fair coin. What is*** $E[X^2]$? ***What is*** $E^2[X]$?

We have the following possibilities of outcome $(h,t)(t,h)(h,h)(t,t)$
Based on that we clearly see that $\Pr(X=0) = \frac{1}{4} \Pr(X=2) = \frac{1}{4}$ *and* $\Pr(X=1) = \frac{1}{2}$
Now we have: $E^2[X] = \left(0 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4}\right)^2 = 1$
$E[x^2] = 0 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} + 4 \cdot \frac{1}{4} = 1.5$

8.4.4 ***An array A of size*** $n > 10$ ***is filled in the following way. For each element*** $A[i]$***, choose two random variables,*** x_i ***and*** y_i ***uniformly and independently from*** $[0,1)$ ***Then set*** $A[i] = \frac{\lfloor 10x_i \rfloor}{10} + \frac{y_i}{n}$
Modify bucket sort so that is sorts the array A in $O(n)$ ***expected time***

The first decimal digit has a uniform random distribution between the elements 0,1,2,3..9
basedon that the ammount of elements expected to fall in each one of them is equal to $\frac{n}{10}$*. However, we must create a number of buckets kind of proportional to the numberof elements*

8.4.5 ***You are given n points in the unit dik*** $p_i = (x_i, y_i,$ ***such that*** $0 < x_i^2 + y_i^2 \leq 1)$ ***for i*** $= 1$ ***to n. Suppose that the points are uniformly distributed, that is, the probability of finding a point in any region of the disk is proportional to the area of the region. Design an algorithm with an average*** $-$ ***case running time of*** $\Theta(n)$ ***to sort the n points by their distances*** $d_i = \sqrt{x_i^2 + y_i^2}$ ***from the origin.***

(***Hint***: ***Design the bucket sizes in Bucket*** $-$ ***Sort to reflect the uniform distribution of the points in the unit disk***)

Since there are a uniform distribution we known that given a random point this point may have As a distance from the origin any value in the interval $(0,1]$ *with equal probability*
Based on that we will have our intervals of bucket sort from 0 *to* n*. so* $n+1$

Equal intervals from $(0,1]$. *We then need to create* $n+1$ *regions inside the circle each one with the same area*

8.4.6 ***A probability distribution function*** $P(x)$ ***for a random variable*** X ***is defined by*** $P(x) = Pr\{X \leq x\}$. ***Suppose that you draw a list of*** n ***random variables*** $X_1 \dots X_n$ ***from a continuou probability distribution*** P ***that is computable in*** $O(1)$***time (given*** y ***you can find*** x ***such that*** $P(x) = y$ ***in*** $O(1)$***time)Givean algorithm that sorts these numbers in linear average*** $-$ ***case time***

We basically need to ensure that we create buckets with equal probability of the items falling in there, with it we can use bucketsort, what grants us Linear time average

---

8.1 ***Probabilistic lower bounds on comparison sorting***
In this problem, you will prove a probabilistic $\Omega(n \lg n)$***lower bound on the running time of any deterministic or randomized comparison sort on*** n ***distinct input elements. You'll begin by examining a deterministic comparison sort*** A ***with decision tree*** T_A.
Assume that every permutation of A***'s inputs is equally likely***

a) Suppose that each leaf of T_A ***is labeled with the probability that it is reached given a random input. Prove that exactly*** $n!$ ***leaves are labeled*** $\frac{1}{n!}$ ***and that the rest are labeled*** 0.

If we are reeiving n *elements, ou algorithm can only output* $n!$ *permutations, if they are equally Likely we have a* $\frac{1}{n!}$ *probability of a given leaf.*

b) Let $D(T)$***Denote the external path length of a decision tree*** T $-$ ***the sum of the depths of all leaves of*** T. ***Let*** T ***be a decision tree with*** $k > 1$ ***leaves, and let*** LT ***and*** RT ***be the left and right subtrees of*** T. ***Show that*** $D(T) = D(LT) + D(RT) + k$
Given the Tree T *it has* k *leaves, lets denote each one of them for* $L_i \dots L_k$
if $k > 1$ *then we have at least* 3 *elements in this tree. For all leaves*
We know that the root is not a leaf, so they are all at the level 1 *from on*
Because of that the leaves of T *are the same as the leaves of the subtrees*
with the difference that now we are on one level lower, so all leaves are now 1 *level lower*
Since we have k *leaves and each one now has a lower level we endup with* $D(T) = D(LT) + D(RT) + k$. *To formalize it we can use summations*:

$$D(T) = \sum_{x \in leaves(T)} height_T(x) = \sum_{x \in leaves(LT)} height_T(x) + \sum_{x \in leaves(RT)} height_T(x)$$

$$= \sum_{x \in leaves(LT)} height_{LT}(x) + 1 + \sum_{x \in leaves(RT)} height_{RT}(x) + 1 \rightarrow$$

$$\sum_{x \in leaves(LT)} height_{LT}(x) + \sum_{x \in leaves(RT)} height_{RT}(x) + \sum_{x \in leaves(LT)} 1 + \sum_{x \in leaves(RT)} 1$$

$$D(LT) + D(RT) + k = D(T)$$

c)Let $d(k)$be the minimun value of $D(T)$over all decision trees T with $k > 1$ leaves.Show that $d(k) = min\{d(i) + d(k-i): 1 \le i \le k-1\}$.
(Hint: Consider a decision tree T with k leaves that achieves the minimiun.
Let $i_0$be the number of leaves in LT and $k - i_0$the number of leaves in RT

Since its a finite set, by the well ordering principle we can guarantee that exists such a minimun
Let $T^*(k)$be the tree such that $D(T^*) \le D(T)$for all Trees with $k > 1$.
Since T^* have k leaves, its LT^* have io leaves and RT^* have $k - io$ leaves
based on that $d(k) = D(T^*) = D(LT^*) + D(RT^*) + k$ And we know that for any other
Tree T we have $D(T) = D(LT) + D(RT) + k$, since $D(T^*)$is the minimun we have:
$D(T^*) \le D(T) \rightarrow D(LT^*) + D(RT^*) \le D(LT) + D(RT)$ for all T
we then want $D(T^*) = \min\{D(LT) + D(RT) + k\}$, But Since LT and RT are independent
In terms of configuration, they are just correlated in terms of leaves. based on that
$\min\{D(LT) + D(RT) + k\} = \min\{d(i) + d(k-i) + k: 1 \le i \le k-1\}$

d) Prove that for a given value of $k > 1$ and i in the range $1 \le i \le k-1$, the function $ilg(i) + (k-i)lg(k-i)$is minimized at $i = \frac{k}{2}$.
Conclude that $d(k) = \Omega(klgk)$

We just need to analyze the derivative. We have that the derivative of the function is zero when $i = \frac{k}{2}$ we then analyze the second derivative and verify that it'sgreater than 0
Implying that when $i = \frac{k}{2}$ we have a minimum

Lets Now prove it by the substitution method that $d(k) = \Omega(klgk)$

Suppose that $cklgk \le d(k)$ for $n_0 \le k < n$. Now lets prove it for n
$d(k) = \min\{d(i) + d(k-i) + k: 1 \le i \le k-1\}$

$d(k) = \min\{d(i) + d(k-i) + k: 1 \le i \le k-1\}\} + k$
$\mathrm{d(k)} \ge \min\{c \cdot ilg(i) + c(k-i)\log(k-i) + k: 1 \le i \le k-1\}\} + k$
$d(k) \ge c \cdot \min\{i \cdot \log(i) + (k-i)\log(k-i) : 1 \le i \le k \le k-1\} + k \rightarrow$
For what we found above we have =
$d(k) \ge 2 \cdot c \cdot \frac{\mathrm{k}}{2}\log\left(\frac{k}{2}\right) + k = c \cdot \mathrm{klog}(k) - ck + k$. *Take $c = 1$ and we are done*
e)Prove that $D(T_A) = \Omega\big(n!\, log(n!)\big)$, and conclude that the average case − time to sort n elements is $\Omega\big(nlog(n)\big)$

Since it has n! leaves and we deduced $d(k)$for k leves we have:
$D(T_A) \ge d(n!) \ge \Omega(n!\log(n!)) \rightarrow$ *then* $D(T_A) = \Omega(n!\log(n!))$
Since we have n! leaves and the total work is $\Omega(n!\log(n!))$. The average work done is
$\Omega(\log(n!)) = \Omega\big(nlog(n)\big)$

Now consider a randomized comparison sort B. We can extend the decision − tree model to handle randomization by incorporating two kinds of nodes: ordinary comparison nodes and randomization **nodes.**
A randomization node models a random choice of the form $RANDOM(1; r)$ made by algorithm B. The node has r children, each of which is equally likely to be chosen during an execution of the algorithm)

***f*)Show that for any randomized comparison sort B, there exists a determinist comparison sort A whose expected number of comparisons is no more than those made by B**

In a randomized node, any path that we take leads us to a leaf and is still valid
Since there are just n children of this node there exist a child with the less average number of comparisons on a path to a leaf. Based on that we can delete all other paths
Because the algorithm will still be right, now this is not randomized but deterministic
Obviously theaverage number of comparisons of this deterministic approach is no larger than the randomized one

---

8.2 *Sorting in place in linear time*
***You have an array of n data records to sort, each with a key of* 0 *or* 1. *An algorithm for sorting such a set of records might posseses some subset of the following three desirable characteristics*:**
1.*The algorithm runs in O(n)time*
2.*The algorithm is stable*
3.*The algorithm sorts in place, using no more than a constant amout of storage space in addition to the original array*

***a*) *Give an algorithm that satisfies criteria* 1 *and* 2:**
***b*) *Given an algorithm that satisfies* 1 *and* 3**
c*) *Give an algorithm that satisfies* 2 *and* 3 *above

a) Counting Sort
b) We can use Counting Sort in a non stable version, so Use C just to count How many Items of each element you have, then proceeds to change A with the amount
c) Inserion Sort

d*) *Can you use any of your sorting algorithms from parts* $(a)-(c)$*as the sorting method used in line* 2 *of radix − Sort, so that Radix − Sort sorts n records with b − bit keys in O(bn) time*? *Explain how or why not

Radix sort needs to use a stable algorithm, Since CountingSort is $O(n+k)$ *and* $k=2$
Counting sort is $O(n)$ *and stable, what allow us to use it in Radix − Sort. We could use the insertion sort on the radix implementation, but it would lead us to* $O(bn^2)$ *average and worst time complexity. About the algorithm I developed at b, since it′snot stable we cannot use it*

e*) *Suppose that the n records have keys in the range* 1 *to k. Show how to modify counting sort so that it sorts the records in place in O(n + k)time.
***You may use O(k)storage outside the input array. Is your algorithm stable*?**

let $C[1:k]$ *be a new array*
for $i=1$ *to* n:
  $C[A[j]]=C[A[j]]+1$
$flag=1$
for $i=1$ *to* k:
  for $L=1$ *to* $C[i]$:
    $A[flag]=i$

$flag = flag + 1$
No this algorithm is not stable

8.3 ***Sorting variable − length items***
a) You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over all the integers in the array is n Show how to sort the array in $O(n)$time.

b) You are given an array of strings, where different strings may have different numbers of characters, but hthe total number of characters over all the strings is n. Show how to sort the strings in $O(n)$time. (The desired order is the standard alphabetical order: for example $a < ab < b$)

a) We can apply a variation of Radix Sort. Say we have k integers in the array, we then Define D as $\max\{d_i, 1 \leq i \leq n\}$ *where d_i indicates the number of digits of the number Array$[i]$. We can* treat *elements with lower digits, say it $D - l$ to have the leftmost l digits being equal* 0, *we don't need to compute it although. We can run Counting sort in each one of this digits. The problem is with dealing with the numbers without The index, because eventhough we are not using it in counting sort, in the standard implementation we would need to move the elements around a n^2 constant factor.*

However if we group elements by number of digits we can apply radix − sort on them and doesn't realize unnecessarily swaps.
We can realize a sort of bucket sort with radix − sort: We iterate over A and based on the number of digits of the element we put it in a Array B.
Whose size is at most n, without any other assumption or information.
To analyze the number of digits of an element we use cd_i $+k$, where k is the constant factor to put it in an appropriate bucket
After that we can apply radix sort in all buckets.
In this situation we will just call Counting Sort a lot of times in some values and all these values combined result in n numbers, so we end up with $O(n)$
Now all elements are sorted in these buckets, since there are n buckets, we iterate over them and concatenate the data, we have m elements and $m < n$, so we still do $O(n)$ work

b) We have basically the same situation, but the sorting order here is a little bit different. We can use Two layers of sppliting this time. First create an array of arrays with 26 *as its size.*
Iterate over the initial array putting in the position 1 *if the first letter is a, position* 2 *if is b and so on. We also keep track of the current position forward that will pass by the counting sort*
This cost us $O(m)$where m is the number of strings. Now for each one of these buckets
We realize the same thing but for the second digit. if a string doesn't have such such digit position we put it in the position of the flag and increment the counter for the counting sort.
With that we realize at most $O(m)$given the $+1$ per string and $O(n)$for all digits since $m < n$, we end up with a $O(n)$running time.

8.4. ***You are given n red and n blue water jugs, all of different shapes and sizes. All the red jugs hold different amounts of water, as do all the blue jugs, and you cannot tell from the size of a jug how much water it holds. Moreover, for every jug of one color, there is a jug of the other color that holds the same amount of water. Your task is to group the jugs into pairs of red and blue jugs that hold the same amount of water.***
To do so, you may perform the following operation: pick a pair of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug.
This operation tells you whether the red jug or the blue jug can hold more water, or that they have the same volume. Assume that such a comparison takes one time unit.
Your goal is to finnd an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.

a) Describe a determinist algorithm that uses $\Theta(n^2)$ comparisons to group the jugs into pairs

Ok so first we are given n red and n blue water jugs and there are n different volumes
We will denote $v(B_i)$ by the volume of a blue water jug at the input index i. the same goest to $v(R_i)$. Now if $i \neq J$ then $v(B_i) \neq v(B_j)$ and for the reds too. for every i of R_i $\exists$ j in B_j such that $v(R_i) = v(B_j)$. Moreover all have diferent shapes. We will denote it by $S(B_i)$ and $S(R_i)$. We can only compare blues with reds. A naive approach would be the following: Take a blue jug, compare with all other red until find the correspondent pair, remove the two repeat the process. The worst case running time is $T(n) = \sum_{i=1}^{n-1} i = O(n^2)$ and best case $O(n)$. Which is better than the required. To worse the approach we can compare all blues with all reds In this situation we have a Tight $\Theta(n^2)$ bound

b) Prove a lower bound of $\Omega(nlg(n))$ for the number of comparisons that an algorithm solving this problem must make.

As shown above the best running time have a $\Theta(n)$ complexity, based on comparison. in some circunstances.

We will generate n pairs so the number of possible outcomes is $n! \cdot \frac{n!}{n!}$
Using a decision tree we verify that we must have n! leaves. So it must have $\log_3(n!)$ depth, what leads to a $\Omega(nlg(n))$. Here we are using a $\log$ in base 3 because our operation of comparison can detemrine in one time unit if its more, less or equal

c) Give a randomized algorithm whose expected number of comparisons is $O(nlgn)$, and prove that this bound is correct.
What is the worst − case number of comparisons for your algorithm?

We can Implement a type of quicksort that uses as pivot for the reds a blue jag and for the blues a red jag:

$JUG - MATCHING(R, B)$
$if(R == 0 \,||B == 0)$:
  $return$
$if(R == 1 \,||B == 1)$:
  $output\ R[1], B[1]$
$else$:
  $B_{==} =$ *random choose a blue jug*
  compare it with all red jugs
  $R_{\ll}$ *is the set of all red jugs lower than the blue one*
  $R_{\gg}$ *is the set of all red jugs greater than the blue one*
  $R_{==}$ *is the one red jug with equal value than the blue one*
  $B_{\ll}$ *is the set of blue jugs lower than the Red jug chosen*
  $B_{\gg}$ *is the set of blue jugs greater then the red jug*
  $output(R_{==}, B_{==})|$
  $JUG - MATCHING(R_{\ll}, B_{\ll})$
  $JUG - MATCHING(R_{\gg}, B_{\gg})$

Now our algorithm becomes a recurrence of type:
$T(n) = T(i) + T(n - i - 1) + \Theta(n)$
This is the same recurrence of quicksort. The worst case happens when
$T(n) = \max\{T(i) + T(n - i - 1)\ 1 \le i \le n - 1\} + \Theta(n)$
This bound is already proven in section 7.4 *Analysis of quicksort*
Then we have that in the worst case $O(n^2)$
The Average case were already analyzed in the quicksort section 7.4 *with a order of* $O(nlg(n))$

8.5 ***Average Sorting***

Suppose that, instead of sorting an array, we just require that the elements increase on average. More precisely, we call an $n -$ ***element array*** A $k -$ ***sorted if, for all*** $i = 1, 2 \ldots n - k$ ***the following holds***:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \le \frac{\sum_{j=i+1}^{i+k} A[j]}{k}$$

a) What does it mean for an array to be $1 -$ ***sorted***?

b) Give a permutation of the numbers $1, 2 \ldots 10$ ***that is*** $2 -$***sorted, but not sorted***

c) Prove that an $n -$ ***element array is*** k ***sorted if and only if*** $A[i] \le A[i + k]$ ***for all*** $i = 1, 2 \ldots n - k$

d) Give an algorithm that $k -$ ***sorts an*** $n -$ ***element array in*** $O\left(nlg\left(\frac{n}{k}\right)\right)$ ***time.***

We can also show a lower bound on the time to produce a $k -$***sorted array, when*** k ***is a constant***

e) Show how to sort a $k -$ ***sorted array of length*** n ***in*** $O(nlg(k))$***time.***
(***Hint: Use the solution to Exercise*** $6.5.11$)

f) Show that when k ***is a constant,*** $k -$ ***sorting an*** $n -$ ***element array requires*** $\Omega(nlgn)$***time.*** (***Hint: Use the solution to part*** (e)***along with the lower bound on comparison sorts***)

$a)\ if\ k = 1\ we\ have: \sum_{j=i}^{i} A[j] \leq \sum_{j=i+1}^{i+1} A[j] \rightarrow A[i] \leq A[i+1]$

$So\ a\ 1-sorted\ array\ is\ just\ a\ sorted\ array$

$b)\ A\ 2-sorted\ array\ is \sum_{j=i}^{i+1} A[j] \leq \sum_{j=i+1}^{i+2} A[j] \rightarrow A[i] + A[i+1] \leq A[i+1] + A[i+2] \rightarrow$

$So\ a\ 2-sorted\ array\ is\ one\ such\ A[i] \leq A[i+2]:$

$as\ an\ example\ we\ have\ 1,6,2,7,3,8,4,9,5,10$

$c) \sum_{j=i}^{i+k-1} A[j] \leq \sum_{j=i+1}^{i+k} A[j] \rightarrow let\ D = j-1.\ then\ if\ j = i+1\ D = i\ and\ when\ j = i+k$

$D = i+k-1 \rightarrow \sum_{j=i}^{i+k-1} A[j] \leq \sum_{D=i}^{i+k-1} A[D+1] \rightarrow \sum_{j=i}^{i+k-1} A[j] - A[j+1] \leq 0$

$telescope\ sum, leaving\ us\ with\ A[i] - A[i+k] \leq 0 \rightarrow A[i] \leq A[i+k]$

$d)\ We\ can\ realize\ the\ following: Divide\ The\ input\ elements\ in\ k\ buckets$
$with\ \left\lfloor \frac{n}{k} \right\rfloor\ items\ in\ each.$
$Now\ we\ realize\ a\ sorting\ algoithm\ in\ each\ one\ of\ these\ buckets.\ Since\ there\ are\ k\ buckets$
$with \frac{n}{k}\ itens\ each\ our\ running\ time\ becomes \frac{n}{k}\log\left(\frac{n}{k}\right) \cdot k = nlog\left(\frac{n}{k}\right)\ so\ sort\ each\ one$
$of\ these\ buckets, now\ we\ can\ iterate\ over\ the\ buckets\ getting\ the\ first\ elements\ of\ them$
$And\ then\ the\ second\ and\ so\ on. This\ wll\ take\ an\ additional\ \Theta(n) time, so\ it's\ still\ nlog\left(\frac{n}{k}\right)$

$e) If\ we\ re\ give\ a\ k-sorted\ array\ we\ can\ create\ k\ sorted\ arrays\ in\ \Theta(n) by\ just$
$iteratingover\ the\ array\ one\ time\ and\ adding\ the\ elements\ in\ the\ right\ buckets.$
$Now\ we\ can\ reuse\ the\ technique\ provided\ by\ merge\ sort. we\ realize \frac{k}{2}\ merge\ in\ the\ beginning$
$\frac{k}{4}\ merges\ after\ and\ so\ on\ until\ 1\ merge. In\ each\ level\ we\ compare\ n\ elements$
$since\ we\ have \log(k)\ levels\ and\ we\ do\ n\ work\ in\ each\ one\ wha\ leads\ us\ to\ O\big(nlg(k)\big)$
$f) Sorting\ each\ part\ is \frac{n}{k}\log\left(\frac{n}{k}\right)\ and\ merging\ is\ \Theta\big(nlog(k)\big), if\ k\ is\ contant\ we\ have$
$\Omega(nlog(n))$

8.6 *Lower Bound on merging sorted lists*

The problem of merging two sorted lists arises frequently. We have seen a procedure for it as the subroutine MERGE in Section 2.3.1. In this problem, you will prove a lower bound of $2n-1$ on the worst-case number of comparisons required to merge two sorted lists, each containing n items. First, you will show a lower bound of $2n - o(n)$ comparisons by using a decision tree.

a) Given 2n numbers, compute the number of possible ways to divide them into two sorted lists, each with n numbers.

b) Using a decision tree and your answer to part (a), show that any algorithm that correctly merges two sorted lists must perform at least $2n - o(n)$ comparisons

Now you will show a lightly tighter $2n-1$ bound.

c) Show that if two elements are consecutive in the sorted order and from different lists, then they must be compared

d) Use you answers to part (c)to show a lower bound of $2n-1$ comparisons for merging two sorted lists

a) Given a list there is only one way to put it sorted way. based on that, the moment that we form A sorted list the other one is already pre − determined. Based on that we have

$2n$ possible numbers to put in n places what leads us to $\frac{2n!}{n!\cdot n!}$

b) All the possibilities shown above reflects all states a sorted list can be divided in.
There is two possible situations, if the maximun of one list is lower than the minimun
We need to perform at maximun $2n$ comparisons because by one comparison w can for sure determine the new element
Now we can verify a decision tree from the leaves to the topall of them when merged must result in the same sorted array. Based on that our decision tree have $\frac{2n!}{n!\cdot n!}$ *leaves*

and the height is atleast h is such $2^h = \frac{2n!}{n!\cdot n!}$, *at least* $\rightarrow h = \log_2 \frac{2n!}{n!\cdot n!}$

This is the amout of operation the algorithm need to correctly merges it $\rightarrow$

$\lg_2 2n! - 2\log(n!) = \sum_{i=1}^{2n} \log(i) - 2\sum_{i=1}^{n} \log(i)$ *. We will now use a lower bound to prove*

We know that $n\cdot lg(n) - \frac{n}{\ln(2)} \leq \sum_{k=1}^{n} lg\ k \leq (n+1)\lg_2(n+1) \rightarrow$

$2n\cdot lg(2n) - \frac{2n}{\ln(2)} \leq \sum_{k=1}^{2n} lg\ k \leq (2n+1)\lg_2(2n+1)$ *and :*

$-2(n+1)\lg_2(n+1) \leq -2\sum_{k=1}^{n} lg\ k \leq 2\left(-n\cdot lg(n) + \frac{n}{\ln(2)}\right) \rightarrow$

$2n\cdot lg(2n) - \frac{2n}{\ln(2)} - (n+1)\lg_2(n+1) + \frac{2n}{\ln(2)} \leq \sum_{k=1}^{2n} lg\ k - 2\sum_{k=1}^{n} lg\ k$

So the number of comparisons is atleast $2n\cdot lg(2n) - 2(n+1)\lg_2(n+1)$

$2nlog(n) + 2n - 2nlog(n+1) - 2\lg(n+1) = 2n + nlog\left(\frac{n}{n+1}\right) - 2\lg(n+1)$

$2n + \log\left(\left(\frac{n}{n+1}\right)^n\right) - \log((n+1)^2) \rightarrow 2n - \left[\log((n+1)^2) - \log\left(\left(\frac{n}{n+1}\right)^n\right)\right] \rightarrow$

$2n - \left[\log\left(\frac{(n+1)^2(n+1)^n}{n^n}\right)\right] \rightarrow$ *so the nmber of comparions are atleast equal to*

$2n - \log\left(\frac{(n+1)^{n+2}}{n^n}\right)$. *Now we just need to prove that* $\log\left(\frac{(n+1)^{n+2}}{n^n}\right)$ *is* $o(n) \rightarrow$

for this, $\forall c > 0$ *we need to find a* n_0 *such that* $\log\left(\frac{(n+1)^{n+2}}{n^n}\right) < cn$. *If* c $>$ *the derivative of the functio on the left we can assure that there will be such* n_0

the derivative is equal to $\frac{1}{1+n} - \log(n) + \log(1+n)$.

The limit when n goes to infinity is 0, *so there is a* n_0 *such that the linear function will have a higher derivative so its just a matterof time for the the linear function be greater for, so the function is* $o(n)$

c)if two elements are consecutive, it means that there is no othter element say it x_j, such that $x_i \leq x_j \leq x_l$, this means that x_i and x_j when compared to any other element from the lists will return the same results, if one is greater, the other will also be greater If one is smallers the other will also be smaller. And we have absolutely no idea the relation between them. If they are not compared we cannot imply any relatio between them

d) each list is n, in the sorted list we hav 2n elements. thats exactly $2n-1$ different consecutive pair, what gives us the minimun number of comparison needed

---

8.7 *The* 0 − 1 *sorting lemma and columnsort*

***A compare − exchange operation on two array elements $A[i]$ and $A[j]$, where $i < j$ has the form*:**

```
COMPARE-EXCHANGE(A, i, j)
1  if A[i] > A[j]
2      exchange A[i] with A[j]
```

After the compare − exchange operation, we know that A[i] ≤ A[j]. An oblivious compare − exchange algorithm operates solely by a sequence of prespecified compare − exchange operations. The indices of the positions compared in the sequence must be determined in advance, and although they can depend on the number of elements being sorted, they cannot depend on the values being sorted, nor can they depend on the result of any prior compare − exchange operation. For example, the COMPARE − EXCHANGE − INSERTION − SORT procedure on the facing page shows a variation of insertion sort as an oblivious compare − exchange algorithm. (Unlike the I NSERTION − SORT procedure on page 19, the oblivious version runs in $\Theta(n^2)$ time in all cases.) The 0 − 1 sorting lemma provides a powerful way to prove that an oblivious compare − exchange algorithm produces a sorted result. It states that if an oblivious compare − exchange algorithm correctly sorts all input sequences consisting of only 0s and 1s, then it correctly sorts all inputs containing arbitrary values

```
COMPARE-EXCHANGE-INSERTION-SORT(A, n)
1  for i = 2 to n
2      for j = i − 1 downto 1
3          COMPARE-EXCHANGE(A, j, j + 1)
```

You will prove the 0 − 1 sorting lemma by proving its contrapositive: if an oblivious compare − exchange algorithm fails to sort an input containing arbitrary values, then it fails to sort some 0 − 1 input. Assume that an oblivious compare − exchange algorithm X fails to correctly sort the array A[1: n]. *Let $A[p]$ be the smallest value in A that algorihm X puts into the wrong location, and let $A[q]$ be the value that algorithm X moves to the location into which $A[p]$ should have gone. Define an array $B[1:n]$ of 0s and 1s as follows*:

$$B[i] = \begin{cases} 0 & \text{if } A[i] \leq A[p], \\ 1 & \text{if } A[i] > A[p]. \end{cases}$$

a)Argue that $A[q] > A[p]$, so that $B[p] = 0$ and $B[q] = 1$

b)To complete the proof of the $0-1$ sorting lemma, prove that algorithm X fails to sort array B correctly

c) Argue that we can treat columnsort as an oblivious compare − exchange algorithm, even if we do not know what sorting method the odd step use

d)Prove that after steps $1-3$ the array consists of clean rows of 0s at the top, clean rows of 1s at the bottom, and at most s dirty rows between them.
(One of the clean rows could be empty)

e) Prove that after step 4, the array, read in column − majhor order, starts with a clean area of 0s, ends with a clean area of 1s, and has a dirty area of at most s^2 elements in the middle. (Again, one of the clean areas could be empty).

f)Prove that steps $5-8$ produce fully sorted $0-1$ output. Conclude that column − sort correctly sorts all inputs containing arbitrary values.

g) Now suppose that s does not divide r. Prove that after steps $1-3$, the array consists of clean rows of 0s at the top, clean rows of 1s at the bottom, and a most $2s-1$ dirty rows between them.
(Once again, one of the clean areas could be empty) How large must r be, compared with s, for columnsort to correctly sort when s does not divide r?

h)Suggest a simple change to step 1 that allows us to maintain the requirement that $r \geq 2s^2$ even when s does not divide r, and prove that with that tyour change, column sort correctly sorts.

a) Since $A[p]$ is in the wrong position there is a element lower than it in a posi tion higher than q or there is an element lower than $A[p]$ in a position lower than p. A $[q]$is salso in the wrong place So both are elements put into the wrong location, but the text says $A[p]$is the lowest then $A[p] < A[q]$, so $B[p] = 0$ and $A[q] = 1$

b)We constructed B in a manner that if $A[i] \leq A[j]$then $B[i] \leq B[j]$. So we follow the same path Since the algorithm puts $A[q]$before $A[p]$ after B being processed it will have $B[q]$lower than $B[p]$, so 1 before 0, what obviously is not sorted

c)Even though we might not use an oblivious compare − exchange algorithm, the result of sorting each column would be the same as if we used an oblivious algorithm

d) Step 1 makes each column have a 0 in the top and 1 in the bottom. Step 2 each group of $\frac{r}{s}$ rows is in row − major order, and it has at most one transition this one transition being 0 to one, The transition of 1 to 0 occur at the end of a group of $\frac{r}{s}$rows, remembering we have s groups of $\frac{r}{s}$rows we dcan have at most s dirty. Step 3 will move the zeros to top and 1 t the bottom The dirties will remain in the middle

e) Step 3 generates a dirty area that is at maximun s rows of depth and s columns wide. Based on the inequalities, its area is at most s^2. During the step 4 we transform the clean on the top to a clean area in the left and the clean 1 in the bottom into clean area in the right. Now the dirty areaare again between them

f)We have at most s^2 values bad. The dirt area after fourth stephas size at most $\frac{r}{2}$

and the following steps will do the sorting. Given that the dirty area has at most $\frac{r}{2}$*, then it resides in one column or in the bottom half of one collumn and top half of the another. Step* 5 *sorts the column containing the dirty area and steps* $6-8$ *maintain the array sorted as whole. In the latter case step* 5 *won't increase, the next steps will move them to the same column and step* 8 *will move it back*

g) In the situation that r is not divisble by s, then after the second step then we can have s transition from 0 *to* 1 *and* $s-1$ *transition from* 1 *to* 0*. after the third step we would have* $2s-1$ *dirty rows, and the area would be* $\leq 2s^2 - s$*. in this case we then need* $r \geq 4s^2 - 2s$

h) Reduce the ammount of transitions in the rows after the second step to at most s by sorting the other columns in reverse order during the first step.