Exercises:

8.1-2
Obtain asymptotically tight bounds on $\lg(n!)$ without using Stirling's approximation. Instead, evaluate the summation $\sum_{k=1}^{n} \lg k$ using techniques from Section A.2.

- Stirling's approximation, used for large n:
  → $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

⇒ By taking log on both sides: $\log(n!) \approx n\log(n) - n\log e$

Side Note
$\log\left(\sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n\right)$
$\log(2\pi n)^{1/2} + \log\left(\frac{n}{e}\right)^n$
$\frac{1}{2}\log(2\pi n)$ $\quad n\log\left(\frac{n}{e}\right)$
if $\log_e = \ln$, $\log e = 1$
$\Rightarrow n(\log(n) - \log e)$
$n(\ln(n) - 1)$
→ And for very large n the term $\frac{1}{2}\log(2\pi n)$ becomes smaller in comparison with

- Evaluating the summation by mathematical induction:

⇒ $\sum_{k=1}^{n} \log(k)$

$\sum_{k=1}^{n} \log(k)$

⇒ Base case: $n = 1$

$\log(1) = 0 \leq c \cdot 0$

⇒ Inductive Hypothesis:

$n = m$

$\log(m!) = c \cdot m\log m$

Inductive Step:

$\log((m+1)!) \leq c(m+1)\log(m+1)$

$\log(m!(m+1)) = \log(m!) + \log(m+1)$

→ $c\,m\log m + \log(m+1) \leq c(m+1)\log(m+1)$

For large n, thus then we have:

$\log((m+1)!) \leq c(m+1)\log(m+1)$ →

Since both cases hold then

$\log(n!) = O(n\log n)$

- By bounding the terms (Maybe for upper bounds)

$\rightarrow \sum_{k=1}^{n} a_k \leq n \cdot a_{max}$, $a_{max} = \{ a_k : 1 \leq k \leq n \}$

$$\sum_{k=0}^{n} \log k \leq n \log n$$

$$\log(n!) \leq n \log n \Rightarrow O(n \log n)$$

- Splitting summations (For lower bound)

$$\sum_{k=1}^{n} \log k = \sum_{k=0}^{n/2} \log k + \sum_{k=n/2+1}^{n} \log k$$

(ignoring $\sum_{k=0}^{n/2} \log k$)

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

$$\geq \frac{n}{2} \log\left(\frac{n}{2}\right) \Rightarrow \frac{n}{2}\left[\log(n) - 1\right]$$

$$\geq (n/2)\log(n) - n/2 \hookrightarrow \Omega(n \log n)$$

- Since $O(n \log n)$ and $\Omega(n \log n)$, we can conclude that $\Theta(n \log n)$.

- Approximation by integrals

For $f(k)$ that are monotonically increasing functions, the following approx. can be done:

$$\int_{m-1}^{n} f(x)\,dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x)\,dx$$

$$\sum_{k=1}^{n} \log k = \int_{1}^{n} \log x\,dx = (x \log x - x)\Big|_{1}^{n}$$

$$= [(n \log n - n) - (0 - 1)]$$

$$= [n \log n - n + 1]$$

Thus, the integral approx.: $\sum_{k=0}^{n} \log k = n \log n - n + 1$

⇒ Following with our inductive step:

$$\sum_{k=0}^{n+1} \log(k) = \sum_{k=0}^{n} \log k + \log(n+1)$$

$$= n\log n - n + 1 + \log(n+1)$$

For n sufficiently large $\log(n+1)$ is approx $\log(n)$

$$\Rightarrow n\log n - n + 1 + \log n$$

$$(n+1)\log(n+1) - (n+1)$$

8.1-3

Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

(1) At least half: $\frac{1}{2} n! \leq 2^n \Rightarrow n! \leq 2^{n+1}$

n	$\downarrow$
0	$1 \leq 2$
1	$1 \leq 4$
2	

- To show it we must understand that from the Theorem, any comparison sort algorith requires $\Theta(n\log n)$. Consider a decision tree of height h and l leaves. Since the input is of size n then the tree comparison tree would have $n!$ leaves. Therefore it average heigh would be $h \geq \log(n!)$, which answer is $\Theta(n\log n)$.

$h=0 \quad n=1$

$1 \quad n=2$

$2^h = n$

$\log(2^h) = \log n$

$h \log(2) = \log(n)$

$h = \log(n)$

- Then this prove that for all inputs it has a tight bound of $\Theta(n\log n)$
- For the problem specifications

$\log\left(\frac{n!}{2}\right) = \log(n!) - \log(2)$, by Stirling approx.

$\Rightarrow n\log n - n\log e - 1$

↳ dictates the growing

$$\log\left(\frac{n!}{n}\right) = \log(n!) - \log(n)$$
$$n\log n - n\log e - \log(n)$$
$$\log\left(\frac{n!}{2^n}\right) = \log(n!) - \log(2^n)$$
$$= n\log n - n\log e - n$$

∴ Which proves that for all of those input sizes the average time would be $\Theta(n\log n)$.

↳ $\log\left(\frac{n!}{n!}\right) = \log(n!) - \log(n!)$
$= 0$

8.1-4

You are given an n-element input sequence, and you know in advance that it is partly sorted in the following sense. Each element initially in position i such that $i \bmod 4 = 0$ 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 position 12 belongs in position 11, 12, or 13. You have no advance information about the other elements, in positions i where $i \bmod 4 \neq 0$. Show that an $\Omega(n \lg n)$ lower bound on comparison-based sorting still holds in this case.

- The problem is trying to ask us if the input sequence is already somehow partially sorted then $\Omega(n\log n)$ still holds?

Divide the problem into two parts:

⇒ For elements at position i partially sorted and $i \bmod 4 = 0$, and at three different position the probability is $3^{n/4}$.

Then for a comparison sort algorithm, this would require:

$$\log(3^{n/4}) = \frac{n}{4}\underbrace{\log(3)}_{\text{constant}} \Rightarrow O(n)$$

- However for the remaining elements, since there is no advance information, then is just like comparing and sorting, then:

$$\log\left(\left(\frac{3}{4}n\right)!\right) = \frac{3}{4}n\log\left(\frac{3}{4}n\right) - \frac{3}{4}n\log e \quad O(n)$$

$= \Omega(n \log n)$

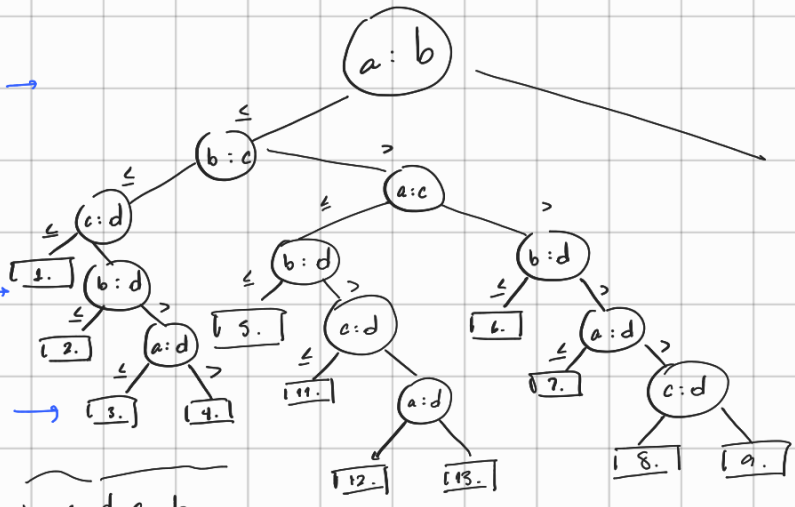
- Please give a merge sort decision tree with four elements *a*, *b*, *c*, and *d*.

- Please give a decision tree for insertion sort operating on four elements *a*, *b*, *c*, and *d*.

For 4 elements there's going to be 24 leafs!

Leaves nodes:

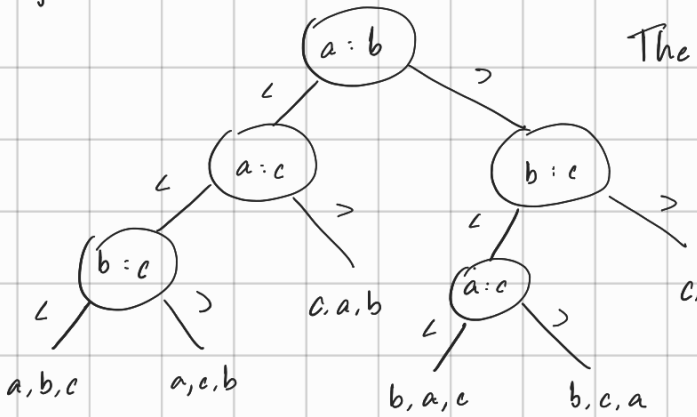
1. $a < b < c < d$
2. $a < b < d < c$
3. $a < d < b < c$
4. $d < a < b < c$
5. $a < c < b < d$
6. $c < a < b < d$
7. $c < a < d < b$
8. $c < d < a < b$
9. $d < c < a < b$
10. $a < c < b < d$
11. $a < c < d < b$
12. $a < d < c < b$



a d c b

↳ Moved on to Merge Sort:

1 a b c
2 a c b
3 c a b
4 b, a, c
5 b, c, a
6 c, b, a



The procedure to solve it is imagine the array a b c the each element is individual and need to be compared to each another

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 the arrays A and C, putting the sorted result back into array A instead of into a new array B.

In-place Counting Sort to be modified:

```
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]
    index = 1
    while i ≤ n :
        correct_position = C[i]
        if A[i] ≠ correct_position
            swap(A[i], correct_position)
            C[i] = C[i] - 1
        else
            index = index + 1
```

8.2-6

Describe an algorithm that, given n integers in the range 0 to k, preprocesses 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 time.

=> Algorithm for Counting sort

```
PREPROCESSING(A, n, k)
for i = 1 to n
    C[A[i]] = C[A[i]] + 1
for i = 1 to k
    C[i] = C[i] + C[i-1]
return C
QUERY(C, a, b)
    if a = 0
        return C[b]
    else
        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 numbers 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 $\Theta(n + 10^d k)$ time.

→ Scale the numbers!

```
COUNTING-SORT-DECIMALS (A, n, k, d)
let B[1:n] and C[0: 10^d * k] be new arrays
for i = 0 to 10^d * k
    C[i] = 0
for i = 1 to n
    A[i] = 10^d * A[i]          → change to integers
    C[A[i]] = C[A[i]] + 1
for i = 1 to 10^d * k
    C[i] = C[i] + C[i-1]        → Cummulative sum
for i = n down to 1
    B[C[A[i]]] = A[i]
    C[A[i]] = C[A[i]] - 1
for i = 1 to n
    B[i] = B[i] / 10^d          // Scale bak to it normal size.
```

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), altogether $2d$ passes over the data occur. Describe how to reduce the total number of passes to $d + 1$.

→ Eliminate the need to run over the second array.

8.3-5

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

Treat the numbers as 3 digit numbers in radix base n, $k = n$. Each digit ranges from 0 to $n-1$. Because in base n the largest 3-digit number is $n^3 - 1$.

There are 3 calls of radix sort each taking $\Theta(n+n)$
$= \Theta(n)$

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 it sorts the array A in $O(n)$ expected time.

→ From the setting of the elements of the array A is made out of two components.

(1) $\frac{\lfloor 10x_i \rfloor}{10}$ which is the primary component

(2) $\frac{y_i}{n}$ which introduce a fractional part to the primary component.

→ Now let's observe that this setting make the elements almost sorted.

The modified Bucket Sort:

```
MOD_BUCKETSORT (A, n)
let B[0:9] be an array of empty lists   // 10 elements due
                                        // to the setting.
for i = 1 to n
    insert A[i] into list B[10*A[i]]
for i = 0 to n-1
    sort list B[i] using Insertion sort  // Insertion sort
concatenate the lists B[0,1...,9]        // is good for
return the concatenated lists.           // almost sorted data
                                         // which average case
                                         // is linear
```

9.1-1

Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. (*Hint:* Also find the smallest element.)

→ To find the smallest element would require $n-1$ comparisons

→ A tournament-like solution where we compare elements in a pairwise manner. The winner is the smallest element and the second smaller the ones that lost against the smaller.

- The smallest has been compared $\lceil \log n \rceil$ rounds in the tournament
- After from the $\lceil \log n \rceil$ that lost against the smaller element we compared them to find the second smaller. Then, $\lceil \log n \rceil - 1$ comparisons.
- Thus, the total number of comparisons is

$$(n-1) + (\lceil \log n \rceil - 1) \Rightarrow n + \lceil \log n \rceil - 2.$$

9.1-3

A racetrack can run races with five horses at a time to determine their relative speeds. For 25 horses, it takes six races to determine the fastest horse, assuming transitivity (see page 1159). What's the minimum number of races it takes to determine the fastest three horses out of 25?

$n+1$ races

→ We need 6 races since at 6th race we make the winners of the past 5 races to compete.

That is $H1_1, H2_1, H3_1, H4_1, H5_1$

Assume that the result of the career is as follow:

$H1_1 > H2_1 > H3_1 > H4_1 > H5_1$

Fastest

Possible candidates:

$H1_2, H1_3$ since this horse might be $> H2_1$ & $H3_1$

$H2_1, H2_2$ since from this group migh be faster

$H3_1$

Conclusion:

→ 5 initial races, more one race for the fastest, more one race for second and third fastest.

Total: 5 + 1 + 1 = 7 races as minimum.

9.2-3

Suppose that RANDOMIZED-SELECT is used to select the minimum element of the array $A = \langle 2, 3, 0, 5, 7, 9, 1, 8, 6, 4 \rangle$. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

→ Select the largest element always, running time $O(n^2)$

9.3-1

In the algorithm SELECT, the input elements are divided into groups of 5. Show that the algorithm works in linear time if the input elements are divided into groups of 7 instead of 5.

9.3-3

Show how to use SELECT as a subroutine to make quicksort run in $O(n \lg n)$ time in the worst case, assuming that all elements are distinct.

```
BEST-CASE-QUICKSORT(A,p,r)
if p < r
   i = ⌊(r-p+1)/2)⌋
   x = SELECT(A,p,r,i)
   q = PARTITION(x)
BC QUICKSORT(A, p, q-1)
BC QUICKSORT(A, q+1, r)
```

9.3-5

Show how to determine the median of a 5-element set using only 6 comparisons.

→ Compare A, B, C, D, E

(1) Compare A and B. Assume A < B

(2) Compare C and D. Assume C < D

(3) Compare B and D. Assume B < D. A is discarded

(4) Compare C and E. Assume C < E

(5) Compare C and D. If D < C then D is the median.
Assume C < D.

(6) Compare C and B. If B < C then B is the median.
otherwise C.