1. *Purpose: reinforce your understanding of the selection problem.*
(6 points) Solve Problem 9.3-1 on page 192 of our textbook.

The algorithm will work in linear time if they are divided into groups of 7.
The number of elements less than(or greater than) the median of the median x will be at least

$$\left\lceil \frac{7}{2} \right\rceil \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{7} \right\rceil \right\rceil - 2 \right) \geq \frac{2n}{7} - 8$$

Therefore, in the worst-case SLECT will be called recursively on at most: n-(2n/7-8)=5n/7+8
And the recurrence is:

$$T(n) \leq T\left(\left\lceil \frac{n}{7} \right\rceil \right) + T\left(\frac{5n}{7} + 8 \right) + \mathrm{O}(n)$$

Which can be shown to be O(n) by substitution(similar to groups of 5 case on Textbook).

With regard to groups of 3,
The number of elements less than(or greater than) the median of the median x will be at least

$$\left\lceil \frac{3}{2} \right\rceil \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{3} \right\rceil \right\rceil - 2 \right) \geq \frac{n}{3} - 4$$

Therefore, in the worst-case SELECT will be called recursively on n–(n/3-4)=2n/3+4
It is therefore the case that

$$T(n) \geq T\left(\frac{n}{3} \right) + T\left(\frac{2n}{3} \right) + \mathrm{O}(n)$$

Page 71 of the text proves that this is O(n lg n) but we need to show $\Omega(n \lg n)$. Note that we get a lower bound if we consider only the part of the tree that is full, which has depth $\lg_3 n$ and each level has cn total cost for some $c > 0$.

We can obtain a lower bound for which k the algorithm will be linear.
Consider the algorithm for group of k.
The number of elements less than(or greater than) the median of the median x will be at least

$$\left\lceil \frac{k}{2} \right\rceil \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{k} \right\rceil \right\rceil - 2 \right) \geq \frac{n}{4} - k$$

Therefore, in the worst-case SELECT will be called recursively on at most: n-(n/4-k)=3n/4+k

The recurrence is $T(n) \leq T\left(\left\lceil \frac{n}{k} \right\rceil \right) + T\left(\frac{3n}{4} + k \right) + \mathrm{O}(n)$

Assume that the algorithm will work in linear time, $T(n) \leq cn$, and let WT(n) be the worst-case running time, then

$$WT(n) = T\left(\left\lceil \frac{n}{k} \right\rceil \right) + T\left(\frac{3n}{4} + k \right) + \mathrm{O}(n) \leq cn$$

Then $T(n) \leq T\left(\left\lceil \frac{n}{k} \right\rceil \right) + T\left(\frac{3n}{4} + k \right) + \mathrm{O}(n)$

$$\le c(\left\lceil \frac{n}{k} \right\rceil) + c(\frac{3n}{4} + k) + \mathrm{O}(n)$$
$$\le c(\frac{n}{k} + 1) + c(\frac{3n}{4} + k) + \mathrm{O}(n)$$
$$= cn(\frac{1}{k} + \frac{3}{4}) + c(1 + k) + \mathrm{O}(n) \quad \text{it is the worst case running time}$$
$$\le cn$$

Since the equation only holds for k>4. So the algorithm for k=3 can not be linear time.

---

2. Purpose: reinforce your understanding of counting sort.

(3 points) You are given an array of numbers A with n elements , your task is to find the largest gap, that is, two numbers A[i] and A[j] such that A[i] < A[j] and there is no A[k] in the array such that A[i] < A[k] < A[j], and so that A[j] – A[i] is maximal. You have to do it in time O(n).

Array of numbers: **A**, Number of elements: **n**

Minimum: **a**, Maximum: **b**, and **g = (b-a)/(n-1)**

Following the hint given, we split the array into **(n-1)** intervals:

[a, a + g)
[a + g, a + 2g)
…
[a + kg, a + (k + 1)g)
…
[a + (n-2)g, a + (n - 1)g]

Note that the intervals are closed on the left-hand side and open on the right hand side, except for the last interval which is closed on both sides, and is equivalent to **[a + (n-2)g, b]**.

Now map each element of the array **A[i]** to the interval **floor((A[i]-a)/g)**, and calculate the minimum and maximum values of each interval as follows:

Let **Max[1..(n-1)]** and **Min[1..(n-1)]** be two arrays of size equal to the number of intervals **(n-1)**
Initialize all **Max[1..(n-1)]** to **-∞**
Initialize all **Min[1..(n-1)]** to **+∞**
For **i** = **1** to **n**
    Let interval index **idx = floor((A[i]-a)/g)**
    If **A[i] > Max[idx]**
        Max[idx] = A[i]
    If **A[i] < Min[idx]**
        Min[idx] = A[i]

Then calculate the differences between consecutive intervals (minimum of the upper interval minus maximum of the lower interval), and find the maximum of these differences. Since we are guaranteed

that the maximum gap does not lie within an interval, and we know that there are no possible values between two consecutive intervals, this maximum will be the maximum gap in the entire array **A**.

```
Initialize MaxGap = -∞
For j = 1 to (n-2)
    Let Gap = Min[j+1] - Max[j]
    If Gap > MaxGap
        MaxGap = Gap
```

Hence, **MaxGap** is the desired max gap. Clearly, since each loop is Θ(n), the entire algorithm is Θ(n).

For those interested, here's a short proof why the maximum gap cannot be less than g (and hence cannot lie within an interval):
Let the max gap be g'
Since there are (n-1) intervals, (n-1)*g' should be at least equal to the length of the range (b-a)
So, we have (n-1)*g' >= (b-a)
Or, g' >= (b-a)/(n-1)
Or, g' >= g

---

3. *Purpose: reinforce understanding of decision trees.*
(4 points) Solve Problem 8-1, parts a–c and e on page 178.

(a) Let NUM be the number of leaves reached in TA
Since no two input permutations can reach the same leaf of the decision tree,
NUM>=n!
Because *A* is a deterministic algorithm, when given a particular permutation as input, it must always reach the same leaf,
So NUM<=n!
Thus, NUM=n!, that is exactly n! leaves are reached, one for each input permutation.
Since each of the n! possible permutations are the input with the probability 1/n!, the n! leaves will each have probability 1/n!, while any remaining leaves have probability 0(because no input will reach these leaves).

b) Let $D(T)$ denote the external path length of a decision tree T ; that is, $D(T)$ is the sum of the depths of all the 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 $D(T) = D(LT) + D(RT) + k$.

Solution:

We know that k>1 and as a consequence of this we can say that the root node is not a leaf node itself. So the root node T will have a left subtree and a right subtree.
To prove $D(T) = D(LT) + D(RT) + k$ we have

$$D(T) = \sum_{x \in leaves(T)} d_T(x)$$

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

$$D(T) = \sum_{x \in leaves(LT)} (d_{LT}(x) + 1) + \sum_{x \in leaves(RT)} (d_{RT}(x) + 1)$$

$$D(T) = \sum_{x \in leaves(LT)} d_{LT}(x) + \sum_{x \in leaves(RT)} d_{RT}(x) + \sum_{x \in leaves(T)} 1$$

hence from the above
$D(T) = D(LT) + D(RT) + k$

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

Solution:

We need to prove $d(k) = \min_{1 \le i \le k-1} \{d(i) + d(k - i) + k\}$.
This can be done in two steps.

i) Show that $d(k) \ge \min_{1 \le i \le k-1} \{d(i) + d(k - i) + k\}$. for this we just need to show that $d(k) \ge d(i) + d(k - i) + k$ for values of i ranging from 1 to k-1. Now we will use the result from part (b) of this problem. If we consider that D(T) is d(k) then let the left sub tree contain i leaves and the right subtree contain k-i leaves. Then by part (b) we have:
$d(k) = D(T)$

= D(RT) + D(LT) + k
= d(i) + d(k-1) + k

since d as minimum of D(T) value, the value changes to
$d(k) \geq d(i) + d(k-1) + k$.
now if either i or k-i are 0, then d would be equal to D(T) which is the opposite to what we said. Hence neither i nor k-i can be 0.

ii) We have to show that $d(k) \leq \min_{1 \leq i \leq k-1} \{d(i) + d(k - i) + k\}$
Similar to the previous one we just have to show that $d(k) \leq \{d(i) + d(k - i) + k\}$ for values of i ranging from 1 to k-i. let us consider that d(k) as D(T), and i as the number of leaves in the left subtree and k-i as the number of leaves in the right sub tree. Then by definition in part (b)
$d(k) \leq D(T)$
= D(RT) + D(LT) + k
= d(i) + d(k-1) + k

From (i) and (ii) we have $d(k) = \min_{1 \leq i \leq k-1} \{d(i) + d(k - i) + k\}$.

---

e) Prove that $D(TA) = \Theta(n!\ \lg(n!))$, and conclude that the expected time to sort n elements is $\Theta(n \lg n)$.

Solution:
We know that $d(k) = \Omega(k \lg k)$
And T_A as n! leaves that are necessary to be considered.
Hence for T_A we can say that:
$D(T_A) \geq d(n!)$
And from the previous step we can expand this further to say
$d(n!) = \Omega(n!\lg(n!))$ → since $d(k) = \Omega(k \lg k)$
$D(T_A)$ denotes the external path length of a decision tree T; that is, $D(T)$ is the sum of the depths of all the leaves of T for all input permutations. This path lengths directly affect the run time of the algorithm. We need to define the total time taken to sort n random numbers. For n! input permutations, each has a probability of 1/n! so for one input permutation, it's the time taken by all permutations by the number of input permutations possible. In this case the number of input permutations possible are n! and the time taken by all are :$\Omega(n!\lg(n!))$ so it is

---

4. *Purpose: Apply recursion to solve a problem, practice formulating and analyzing algorithms.* In the US, coins are minted with denominations of 50, 25, 10, 5, and 1 cent. An algorithm for making change using the smallest possible number of coins repeatedly returns the biggest coin smaller than the amount to be changed until it is zero. For example, 17 cents will result in the series 10 cents, 5 cents, 1 cent, and 1 cent.
a) (2 points) Write a recursive algorithm for changing *n* cents.
b) (2 points) Analyze its time complexity.
c) (2 points) Write an O(1) (non-recursive!) algorithm to compute the number of returned coins.
d) (1 point) Show that the above greedy algorithm does not always give the minimum number of coins in a country whose denominations are 1, 6, and 10 cents.
e) (2 points) What is the average number of coins that your algorithm from part (a) returns if you have to change 1,2,…,100 cents, and every amount appears with the same probability?

(1)Recursive algorithm for changing n cents and m1,m2,m3,m4,m5 stand for number of 50,25,10,5,1 cent respectively.

```
Main(n)
0 ← m1
0 ← m2
0 ← m3
0 ← m4
0 ← m5
Change(n,m1,m2,m3,m4,m5)


Change(n,m1,m2,m3,m4,m5)
if n>=50
   n ← n − 50
   m1 ← m1 + 1
else if n>=25
```

$n \leftarrow n-25$

$m2 \leftarrow m2+1$

else if n>=10

$n \leftarrow n-10$

$m3 \leftarrow m3+1$

else if n>=5

$n \leftarrow n-5$

$m4 \leftarrow m4+1$

else if n>=1

$n \leftarrow n-1$

$m5 \leftarrow m5+1$

endif

If n>0

Change(n,m1,m2,m3,m4,m5)

else

return m1,m2,m3,m4,m5

endif

(2) Time complexity.

The steps for counting numbers of 25,10,5,1 cents($m2 \leftarrow m2+1$, $m3 \leftarrow m3+1$, $m4 \leftarrow m4+1, m5 \leftarrow m5+1$) are all constant, for instance, whatever the n is, the number of 25 cents will no more than 1, and 10 cents will no more than 2, 5cents maximum 1, 1 cents 4.

But the step for counting 50 cents is dependent on n. The running time is $\Theta(\frac{n}{50}) = \Theta(n)$

(3) Non-Recursive algorithm

Change1(n,m1,m2,m3,m4,m5)

$0 \leftarrow m1$

$0 \leftarrow m2$

$0 \leftarrow m3$

$0 \leftarrow m4$

$0 \leftarrow m5$

if n>0

$m1 \leftarrow n/50$

$n \leftarrow n\%50$

$m2 \leftarrow n/25$

$n \leftarrow n\%25$

$m3 \leftarrow n/10$

$n \leftarrow n\%10$

$m4 \leftarrow n/5$

$n \leftarrow n\%5$

(4, i.e., part d) All that is needed is a simple counterexample. Consider the value 12. A greedy approach would first use the coin valued at 10 and then 2 value 1 coins for a total of 3. The optimum solution uses only two coins of value 6.

(5, i.e., part e) We want to know the average number of coins for each value from 1 to 100. Since all amounts are equally probable it suffices to get the total and divide by 100. We include $n = 0$, to make the grouping easier to see. We look at values in groups of 5, starting with 0 – 4, where it's all pennies and the total number of coins used is 10.

Groups of 5 where each value uses one coin more than those of 0 – 4 are

5 - 9,
10 - 14,
25 - 29

each group totaling 15 (add a single coin to each of 5 values)

for other groups less than 50, we get

15 - 19,
20 - 24,
30 - 34,
35 - 39

each totaling 20 (add 10+5, 10+10, 25+5, and 25+10, respectively) and

40 - 44
45 - 49

each totaling 25 (add 25+10+5 and 25+10+10, respectively)

Now, for amounts < 50, we have 10 + (3 * 15) + (4 * 20) + (2 * 25) = 10 + 45 + 80 + 50 = 185

Each amount ≥ 50 (other than 100) adds a coin to a value < 50, so we get a total of 235 for those. Finally there are 2 coins to make 100.

Overall total = 185 + 235 + 2 = 422.

Thus, the average is 4.22