Problem 1

1. (a) Since we are trying to prove that $f(n) = O(n^2)$, we need to show that $f(n) \leq kn^2$ for large n and some choice of k. Plug the expression kn^2 into the recurrence relation:

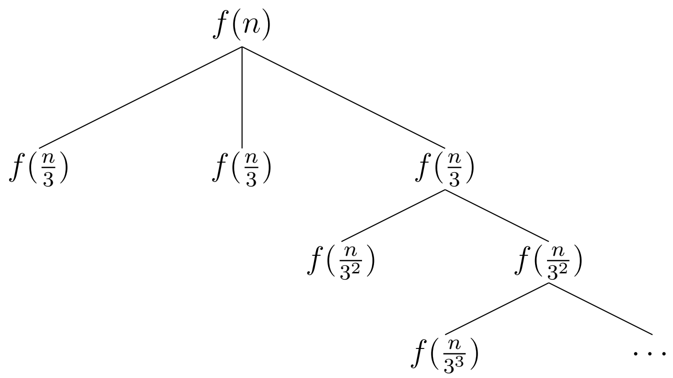
$$16f(\frac{n}{4}) + 514n \leq 16k(\frac{n}{4})^2 + 514n \leq kn^2 \tag{1}$$

However, we can see that the $514n$ is hard to eliminate. Therefore, by adding a constant behind, we can remove it without conflict with the assumption that $f(n) \leq kn^2$. The new equation is $f(n) = kn^2 - dn$, plug this into the relationship:

$$16f(\frac{n}{4}) + 514n \leq 16(k(\frac{n}{4})^2 - dn) + 514n \leq kn^2 - dn \tag{2}$$

After rearranging the elements, we can see that when $d \geq \frac{514}{15}$, d can makes this inequality come true for sufficiently large n. Therefore, this function is $O(n^2)$.

(b) The equation from the statement is $f(n) = 27f(\frac{n}{3}) + 40e^3n^3$, we can know that each child level has 27 items. Which can be portraited as:



From the denominator of the fractions, we know that the depth of this tree is log_3n, which also means thhere are log_3n leaves at the bottom. Knowing how many leaves we have, the equation can rewrite into:

$$\begin{aligned} 27f(\frac{n}{3}) + 40e^3n^3 &= 40e^3n^3 + 40e^3n^3 + \cdots + 40e^3n^3 \\ &= 40e^3n^3 * log_3n \\ &= O(n^3logn) \end{aligned} \tag{3}$$

2. The problem requires us to categorize all the equations, so I've enlisted them into a table. All of them are estimated by recursion tree, since proving all of them is too time consuming.

Classes	Functions
$\Theta(1)$	$n^{\frac{1}{lgn}}$, 2147483647, 2^{10000}
$\Theta(log(log(n)))$	$lg(ln(n))$
$\Theta(log(n))$	$f(n) = f(n-1) + \sum_1^n \frac{1}{i}$
$\Theta(n^{log\sqrt{2}})$	$(\sqrt{2})^{logn}$
$\Theta(\frac{n}{log(n)})$	$\frac{n}{ln(n)}$
$\Theta(n)$	$e^{ln(n)}$, $(\frac{10}{e})n$
$\Theta(nlog(log(n)))$	$f(n) = \sqrt{n}f(\sqrt{n}) + n$, $enlg(ln(n))$
$\Theta(nlog(n))$	$nlog(n)$, $ln(n!)$, $nlg(n)$
$\Theta(n^{\frac{3}{2}})$	$n^{\frac{3}{2}}$
$\Theta(n^{log(e)})$	$f(n) = ef(\frac{n}{2})$
$\Theta(n^3)$	$e^5n^3 - 10n^2 + e^1000$
$\Theta(n^{e+1})$	$f(n) = f(n-1) + n^e$
$\Theta(n^{log(log(n))})$	$(lg(n))^{ln(n)}$, $nlg^{ln(n)}$, $n^{lg(lg(n))}$
$\Theta((\frac{1+\sqrt{5}}{2})^n)$	$f(n) = f(n-1) + f(n-2)$
$\Theta(n!)$	n!

Problem 2

1. (a) The problem clearly states that $O(nlogn)$ is the required complexity. Which hints us that divide and conquer is needed here. Because

$$T(n) = 2T(\frac{n}{2}) + O(n) \tag{4}$$

is the complexity for a simple "two-half" way of dividing, and it has the complexity of $O(nlogn)$.

function FIND-MAJORITY(array)
  if $array.size() = 1$ **then**
    return array[1]
  end if
  $k \leftarrow \lfloor \frac{n}{2} \rfloor$
  $element_{left} \leftarrow$ FIND-MAJORITY($array[1..k]$)
  $element_{right} \leftarrow$ FIND-MAJORITY($array[k+1..array.size()]$)
  if $element_{left} = element_{right}$ **then**
    return $element_{left}$
  end if
  $count_{left} \leftarrow$ FREQUENCY($array, element_{left}$)
  $count_{right} \leftarrow$ FREQUENCY($array, element_{right}$)
  if $count_{left} > k+1$ **then**
    return $element_{left}$
  else if $count_{right} > k+1$ **then**
    return $element_{right}$
  else
    return -1
  end if
end function

The function **frequency** is used to check for the frequency of appearance of the majority elements, in order to determine which half of the array acutally contains the majority element after they combined. During the return statements, -1 indicates no majority elements is founded.

References

- `http://www.ece.northwestern.edu/~dda902/336/hw4-sol.pdf`

(b) Since a majority element in an array of size n, is an element tha appears more than $\lfloor \frac{n}{2} \rfloor$ times. We can use Moore's voting algorithm, which compose of two steps.

1. Get an element occuring most of the time in the array.
2. Check if the element obtained from above step is majority element.

The basic idea of Moore's voting alogithm is that: if we cancel out each occurence of an element with all the other elmements that are different from it, then it will exist till the end if it's a majority element.

```
function FIND-CANDIDATE(array)
    count ← 0
    for i < array.size() do
        if count = 0 then
            elements ← array[i]
        else if array[i] = element then
            count ← count + 1
        else
            count ← count − 1
        end if
    end for
    count ← 0
    for i < array.size() do
        if array[i] = element then
            count ← count + 1
        end if
    end for
    if count > array.size()/2 then
        return element
    else
        return -1
    end if
end function
```

After the candidate is found, step 1 is completed, which shall cost $O(n)$ only. By definition, it has to appears more than $\lfloor \frac{n}{2} \rfloor$ times, we verify it in this step. (Notes that -1 indicates no majority elements is founded as well.)

```
function VERIFY-MAJORITY(array, size)
```

$candidate \leftarrow$ FIND-CANDIDATE$(array, size)$
$count \leftarrow 0$
for all elements in array **do**
  if $element = candidate$ **then**
    $count \leftarrow count + 1$
  end if
end for
if $count \geqslant \lfloor \frac{n}{2} \rfloor$ **then**
  return candidate
else
  return -1
end if
end function

As for the complexity, it's not hard to see that $T(n) = T(\frac{n}{2}) + O(n)$ will reach to $O(n)$ in this case, therefore the algorithm satisfies the requirement.

References

- `http://www.geeksforgeeks.org/majority-element/`

2. (a) If two arrays, each has the size of i and j, than merging them will cost $O(i+j)$, which is literally linear time $O(n)$. Knowing the fact above, now we have k arrays, and each of size n, merging them is:

$$(n+n) + (2n+n) + \cdots + ((k-1)n + n) = \frac{(k-1) \times k}{2} n + kn \tag{5}$$

The result can be classified as $O(k^2 n)$.

(b) If we merge them two at a time, through divide and conquer, the array pile can rewrite in a recursive form:

$$T(n) = T(\frac{k}{2}) + O(2n) = T(\frac{n}{2}) + O(n) \to O(knlogn) \tag{6}$$

When the algorithm reaches the leaves, it only needs to perform $n + n$ times of merging for the arrays, since each of the atomic array are of size n, which is the source of $O(2n)$.

Problem 3

1. (a) We first take the labeling that will comprise of minimal total length of the segments, this is obviously exsists, since there are only finitely many labelings. If $\overline{A_iB_i}$ intersects $\overline{A_jB_j}$ at X, then $\overline{A_iB_j} < \overline{A_iX} + \overline{XB_j}$, $\overline{A_jB_i} < \overline{A_jX} + \overline{XB_i}$ (by the triangle inequality). So $\overline{A_iB_j} + \overline{A_jB_i} < \overline{A_iB_i} + \overline{A_jB_j}$. So the labeling was not minimal, which contradicts to our previous assumption. Hence, no pair of segments can intersect.

References

- 40th Putnam 1979, A4 solution. `https://goo.gl/q8L4gr`

(b) According to the paper from Chi-Yuan Lo, we can have a ham sandwich cut in $O(n)$.

The input is an interval $T = [\,l, r\,]$ on the x-axis, sets G_1 and G_2 of black and white lines ($|G_1| \geq |G_2|$) and integers p_1 and p_2 that indicate the p-levels in G_1 and G_2 corresponding to the median levels of the original sets.

Steps are the following:

i. Divide the interval T into a constant number of subintervals $T_1, \cdots,$ $T_C(C \geq \frac{1}{\alpha})$ such that no $V(T_i)$ contains more than a prescribed (constant) fraction of the vertices of the arrangement of $G1[\,O(m1)]$.
ii. Find one subinterval T_i with the odd intersection property $[\,O(m_1 + m_2)]$.
iii. Construct a trapezoid $\tau \in V(T_i)$, such that
   - $\lambda_1 \cap V(T_i) \subset$
   - At most half of the lines of G_1 intersect $\tau[\,O(m_1)]$
iv. Discard all the lines of G_1 which do not intersect τ (at least $\frac{m_1}{2} > \frac{m_1+m_2}{4}$ lines), and update p_1 accordingly ($p_1' \leftarrow p_1 - b$, b denoting the number of discarded lines of G_1 lying completely below τ). Then T_i becomes the new T, and we are ready for the next phase of the algorithm (i.e. recursive call, swaping G_1' and G_2' if $|G_1| < |G_2|$) $[\,O(m_1 + m_2)]$.

For more illustration for better comprehension of the algorithm, please reference `http://goo.gl/Gcf9Lm` *for the detail picture and JavaScript demonstration.*

References

- Algorithms for Ham-Sandwich Cuts. Chi-Yuan Lo. Discrete and Computational Geometry. 1994.
- Step To Find the Ham Sandwich Cut. `http://goo.gl/700sYo`
- Ham-Sandwich Cuts. `http://goo.gl/Gcf9Lm`

(c) *I'm using the result from (2), presumably I use the hint from (2), though I'm using a different approach in (2), and has a far better time complexity.*

From previous proof and the reference, we know that there *must* exists a cut that can perfectly separate the result, therefore, we first randomly choose a dot in the set. The randomly choosed dot will now act as the origin, while the Cartesian plane itself doesn't rotate.

Now we sort the dots according to their absolute angle with the x-axis. Which will be the main bottle neck of this algorithm.

function FIND-CUT(origin, $dots[1 \cdots n]$)
$dots \leftarrow$ SORT($origin$) $counter \leftarrow 0$ ▷ Sort relative to the angle between the x-axis, CCW.

for all $i = dots[1 \cdots n]$ **do**
    if i is black **then**
        $counter \leftarrow counter + 1$
    else
        $counter \leftarrow counter - 1$
    end if
    if counter = -1 **then**
        break
    end if
end for
return i
end function

The algorithm returns the connected dot, when the origin and the connected dot are connected, forming the line L, it will satisfy the condition. To perform the ham sandwich cut, shift the line CW, with the returned dot as the pivot, and voilà!

The time spend is determined by the sorting algorithm, which shall locates at $O(nlogn)$ in a general condition. Consider the worst-case-scenario, which means that the randomly choosed dot always leads to consecutive dots in the plane, than we shall move on to other dots, since every dot is paired with the other, it won't matter which one we pick first. In this case, we already sorted them, meaning that only $\frac{n}{2}$ of dots need to try out. Hence, in the worst-case-scenario, $O(nlogn)$ has to multiply $\frac{n}{2}$, leading to a time complexity of $O(n^2logn)$.