Name: Chloe Hu
Student ID: 21044009
Email: chuap@connect.ust.hk

1.

a) A = Ω(B)
b) A = Θ(B)
c) A = O(B)
d) A = Ω(B)
e) A = Ω(B)
f) A = Ω(B)
g) A = Ω(B)

2.

(a) $T(1) = 1$; $T(n) = 4T(n/2) + n^2$ for $n > 1$.

$$T(n) \quad n^2 \longrightarrow n^2$$

$$T\left(\tfrac{n}{2}\right)\ T\left(\tfrac{n}{2}\right)\ T\left(\tfrac{n}{2}\right)\ T\left(\tfrac{n}{2}\right) \quad \left(\tfrac{n}{2}\right)^2 \text{ each} \longrightarrow 4 \times \frac{n^2}{2^2} = n^2$$

$$\vdots \quad T\left(\frac{n}{2^k}\right) \longrightarrow 4^i \times \left(\frac{n}{2^i}\right)^2 = n^2$$

$$1 \quad T(1) \quad \cdots \quad T(1) \quad 1 \longrightarrow 4^{\log_2 n} = n^{\log_2 4} = n^2$$

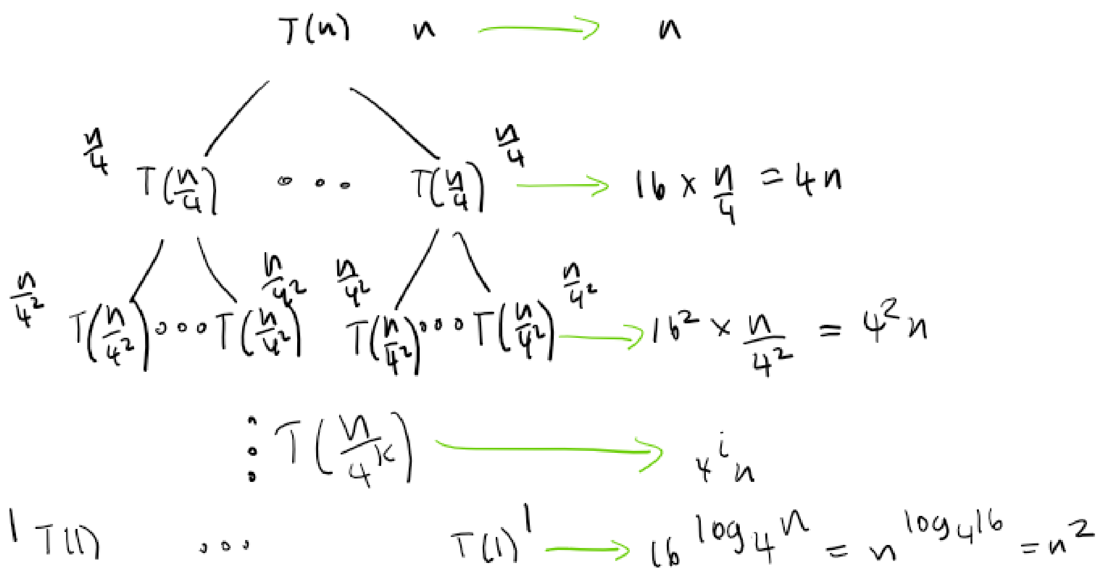
$$T(n) \le n^2 + \sum_{i=0}^{\log_2 n - 1} n^2$$

$$= n^2 + (\log_2 n - 1 - 0 + 1) n^2$$

$$= n^2 + n^2 \log_2 n$$

$$T(n) = O(n^2 \log_2 n)$$

(b) $T(1) = 1$; $T(n) = 16T(n/4) + n$ for $n > 1$.



$T(n)$ n → n
$\frac{n}{4}$ $T(\frac{n}{4})$ ... $T(\frac{n}{4})$ $\frac{n}{4}$ → $16 \times \frac{n}{4} = 4n$
$\frac{n}{4^2}$ $T(\frac{n}{4^2})$ ... $T(\frac{n}{4^2})$ $\frac{n}{4^2}$ $\frac{n}{4^2}$ $T(\frac{n}{4^2})$ ... $T(\frac{n}{4^2})$ $\frac{n}{4^2}$ → $16^2 \times \frac{n}{4^2} = 4^2 n$
$T(\frac{n}{4^k})$ → $4^i n$
1 $T(1)$ ... $T(1)$ 1 → $16^{\log_4 n} = n^{\log_4 16} = n^2$

$$T(n) \le n^2 + n \sum_{i=0}^{\log_4 n - 1} 4^i$$

$$= n^2 + n \left(\frac{4^{\log_4 n} - 1}{4 - 1} \right)$$

$$= n^2 + n \left(\frac{n-1}{3} \right)$$

$$T(n) = O(n^2)$$

(c) $T(2) = 1$; $T(n) = T(\sqrt{n}) + 1$ for $n > 1$.

$$T(n) = T(\sqrt{n}) + 1 = T(n^{\frac{1}{2}}) + 1$$
$$= T(n^{\frac{1}{4}}) + 1 + 1 = T(n^{\frac{1}{4}}) + 2$$
$$= T(n^{\frac{1}{8}} + 1 + 2 = T(n^{\frac{1}{8}}) + 3$$
$$= T(n^{\frac{1}{2^i}}) + i$$

finding levels

$$n^{\frac{1}{2^i}} = 2$$
$$\frac{1}{2^i} \log_2 n = 1$$
$$\log_2 n = 2^i$$
$$\log_2(\log_2 n) = i$$

$$a^b = c$$
$$\log_a c = b$$

$$T(n) \leq 1 + \log_2(\log_2 n)$$
$$T(n) = \log(\log n)$$

3.
  a)

Description:
The recursive algorithm has a base case when n = 1. To get to n = 1, the algorithm recursively calls itself with an input of (n - 1) each time. It adds a [0] or [1] to the current n x n binary array to create all the possible configurations of the binary array. Once the current array has gone through n^2 calls (ie. all n x n cells have been filled up) it is pushed to completedArrays. This ensures that all combinations of the binary array are created.

Pseudocode:

```
function createBinaryArrays(n, currArray=[], completedArrays=[], rows=0, cols=0)
    // we have filled all rows and cols of the binary n x n array
    if rows = n
        completedArrays.push(currArray)
    // we have filled an entire row, so increment to the next row
    else if cols = n then
        createBinaryArrays(n, currArray, completedArrays=[], rows + 1, 0)
    // recursively fill the next col with 0
    else
        createBinaryArrays(n, currArray + [0], completedArrays=[], rows, cols + 1)
    // recursively fill the next col with 1
        createBinaryArrays(n, currArray + [1], completedArrays=[], rows, cols + 1)
    return solutions
```

b)

Let T(n) be the number of iterations needed for an input size of n.

The base case is when n = 1, in this case we add [0] or [1] to the currArray. Thus O(1) work is done.

The algorithm recursively calls itself with an input size of (n - 1) splitting 2 times each call as it adds [0] or [1] to the currArray. As a result the algorithm performs $2T(n - 1)$ recursive calls to iterate through the columns of each row.

Since there are n rows, it repeats the aforementioned recursion n times. As a result the algorithm performs $2T(n - 1) \times T(n)$ amount of work to generate all binary arrays.

For simplicity, $2T(n - 1) \times T(n)$ was simplified into $3T(n - 1)$ in the analysis below.

$$T(1) = 1 \; ; \; T(n) = 2T(n-1)\,T(n) + 1$$
$$\approx 3T(n-1) + 1$$
$$= 3(3T(n-2)+1) + 1 = 9T(n-2) + 3 + 1$$
$$= 9(3T(n-3)+1) + 3 + 1 = 27T(n-3) + 3^2 + 3^1 + 3^0$$
$$= 3^i T(n-i) + 3^{i-1} + 3^{i-2} + \ldots + 3^0$$

$$n - i = 1 \Rightarrow i = n-1$$

$$T(n) = 3^{n-1} + \sum_{i=0}^{n-1-1} 3^i$$
$$= 3^{n-1} + \frac{3^{n-2+1} - 1}{3-1}$$
$$= 3^{n-1} + \frac{3^{n-1} - 1}{2}$$
$$T(n) = O(3^n)$$

4.

a) The maximum number of 10-major elements in A is 9. For a number a 10-major element it must occur more than $\frac{n}{10}$. Thus:

$b > \frac{n}{10}$, where b is the number of times a number appears in A.

As a result, the minimum value that b can be is $\frac{n}{10} + 1$, rounded to the nearest integer. To find how many 10-major elements, we compute $\frac{n}{b}$. Simplifying gives $\frac{n}{n/10+1}$.

The function $\frac{n}{n/10+1}$ has an asymptote at 10, thus the maximum number of 10-major is the floor of the function. As a result, the maximum number of 10-major elements is 9.

b)

Description:
The base case is when the number of elements in the array (n) is less than 10. This is because a single occurrence in an array of size 9 will be a 10-major (1 > 9/10). Once the subarray has size 9, it reports the unique elements in the subarray.

If the subarray has size greater than or equal to 10, the algorithm splits into left and right halves, recursively finding the 10-major element in each half and adding the 10 major elements to the leftMajors and rightMajors array.

Then the algorithm removes the duplicates from the two arrays and puts all the unique values into the possibleMajors array.

Finally, the algorithm checks if the values in the possibleMajors array are truly 10-major by counting how many times they occur in the given array (A). This needs to be done because a 10-major in the subarray is not necessarily a 10-major in the given array.

Pseudocode:

```
function findTenMajor(A, start, end)
    // base case
    if (end - start) < 10
        return unique numbers

    middle = floor((end + start)/2)

    //recursively find all the possible 10-majors using binary search in left and right portions of the
    array
    leftMajors = findTenMajor(A, start, middle)
    rightMajors =  findTenMajor(A, middle + 1, end)

    // remove duplicates
    possibleMajors = leftMajors ∪ rightMajors

    actualMajors = []

    // check possibleMajors to see if they are truly 10-major in A
    for all nums of possibleMajors
        count = 0
        for k = start to end
            if A[k] = nums
                count++
        if count > end/10
            actualMajors.push(x)

    return actualMajors or “No majors found”
```

c)

Let T(n) be the number of iterations needed for an input size of n.

Since the maximum number of distinct 10-major numbers in the array is 9, if n = 9, then O(1) work is performed. Thus T(9) = 1.

When n > 9, the algorithm recursively calls itself on the left and right halves of the array, splitting the number of items in the array by 2 each time (n/2). Thus T(n/2) work is performed for both the left and right recursions.

To find the union of leftMajors and rightMajors O(1) time would be needed. This is because the maximum number of 10-major items in each array is 9. Thus the time to perform the union is $O(|leftMajors| + 1)(|rightMajors) + 1) = O(100) = O(1)$

Finally the algorithm checks to see if the possibleMajor elements are 10-major elements which takes O(n) time. This is because the algorithm must loop through all the possible 10-major elements and count how many times the number occurs. In the worst case, it would cause *n* iterations.

Thus $T(9) = 1$ and

$$T(n) \leq T(\frac{n}{2}) + T(\frac{n}{2}) + O(n).$$

d)

$T(9) = 1$; $T(n) \leq 2T(\frac{n}{2}) + n$

$= 2\left(2T(\frac{n}{4}) + n\right) + n = 4T(\frac{n}{4}) + 2n + n$

$= 4\left(2T(\frac{n}{8}) + n\right) + 2n + n = 8T(\frac{n}{8}) + 4n + 2n + n$

$= iT(\frac{n}{2^i}) + n\left(2^0 + 2^1 + 2^2 + \ldots 2^{i-1}\right)$

$\frac{n}{2^i} = 9 \Rightarrow 2^i = \frac{n}{9} \Rightarrow i = \log_2(\frac{n}{9})$

$T(n) = \log_2(\frac{n}{9}) + n \sum_{i=0}^{\log_2(\frac{n}{9})-1} 2^i$

$= \log_2(\frac{n}{9}) + n\left(\frac{\log_2(\frac{n}{9}) - 1 + 1 - 1}{2-1}\right)$

$= \log_2(\frac{n}{9}) + n\left(\log_2(\frac{n}{9}) - 1\right)$

$= \log_2(\frac{n}{9}) + n\log_2(\frac{n}{9}) - n$

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