Here are some extra practice problems taken from past semesters. You are welcomed to check your solutions with me or discuss them in the telegram group chat.

1. **Asymptotic Analysis** (AY 22/23 Sem 2). State with proof whether the following statements are TRUE or FALSE.

   (a) Suppose $f(n) = n(1 - \sin n)$ (defined over non-negative reals). Then $f(n) \notin O(n)$.

   (b) Suppose $f(n) = n(1 - \sin n)$ (defined over non-negative reals). Then $f(n) \notin \Omega(n)$.

   (c) Let $T(n) = 7T(n/7) + f(n)$ (with the base case being $T(1) = \Theta(1)$). If $f(n) \in O(n \log n)$, then $T(n) \in \Omega(n \log n)$.

   Solution:

   (a) FALSE. Since $\sin n \geq -1$, we have $f(n) = n(1 - \sin n) \leq 2n$. Hence, $f(n) \in O(n)$.

   (b) TRUE. For any n_0 and c, take $n = 2z\pi + \pi/2$. Since $\sin n = 1$, we have $f(n) = n(1 - \sin n) = 0 < c \cdot n$. Hence, $f(n) \notin \Omega(n)$.

   (c) FALSE. Consider $f(n) = n^{1/3} \in O(n \log n)$. Since $f(n) = O(n^{1-\varepsilon})$ for $\varepsilon = 2/3$, we get $T(n) = \Theta(n)$ by case 1 of the master theorem. Hence, $T(n) \notin \Omega(n \log n)$.

2. **Recurrences** (AY 24/25 Sem 2). Solve the following recurrence relations. Provide as tight a bound as possible. If you use the master theorem, state the case that applies, along with a short reasoning why the case applies. Otherwise, give a detailed proof (using any of the methods). Unless otherwise specified, you can assume base cases, $T(n)$ for $n \leq$ some constant, to be $\Theta(1)$.

   (a) $T(n) = 6T(n/3) + n^2$.

   (b) $T(n) = 9T(n/2) + 6n^3 + 4$.

   (c) $T(n) = T(n/7) + T(6n/7) + 5$. (Assume $7 \leq T(n) \leq 100$ for $n \leq 7$.)

   Solution:

   (a) Using master theorem, Case 3: $f(n) = n^2 = \Omega(n^{log_3 6 + \varepsilon})$, where $\varepsilon = 2 - \log_3 6 > 0$. Furthermore, regularity condition is satisfied, as $6f(n/3) = 6(n/3)^2 = 6n^2/9 \leq (6/9)f(n)$. Thus, $T(n) = \Theta(n^2)$.

   (b) Using master theorem, Case 1: $f(n) = 6n^3 + 4 = O(n^{\log_2 9 - \varepsilon})$, where $\varepsilon = \log_2 9 - 3 > 0$. Thus, $T(n) = \Theta(n^{\log_2 9})$.

   (c) Use substitution method.
   (Lower bound) Want to show: $T(n) \geq cn$ where $c = 1$.
   Basis Step: For $n \leq 7$, $T(n) \geq 7 \geq cn$.
   Induction Step:

   $$\begin{aligned} T(n) &= T(n/7) + T(6n/7) + 5 \\ &\geq cn/7 + c6n/7 + 5 && \blacktriangleleft \text{ by induction hypothesis} \\ &= cn + 5 \geq cn \end{aligned}$$

   (Upper bound) Want to show: $T(n) \leq c_1 n - c_2$ where $c_1 = 105$, $c_2 = 5$.
   Basis Step: For $n \leq 7$, $T(n) \leq 100 \leq c_1 n - c_2$.
   Induction Step:

   $$\begin{aligned} T(n) &= T(n/7) + T(6n/7) + 5 \\ &\leq c_1 n/7 - c_2 + 6c_1 n/7 - c_2 + 5 && \blacktriangleleft \text{ by induction hypothesis} \\ &= c_1 n - 2c_2 + 5 \\ &\leq c_1 n - c_2 && \blacktriangleleft \text{ since } c_2 \geq 5 \end{aligned}$$

3. **Recurrences** (AY 23/24 Sem 2 Practice Set). For each of these, try figuring out why you can't use master theorem to solve these recurrences, you do not need to be formal. Next, get (with proof) as tight bound as possible, for both upper and lower.

   (a) $T(n) = 2T(n-1) + \Theta(1)$

   (b) $T(n) = T(\sqrt{n}) + \Theta(1)$

   (c) $T(n) = 2T(\sqrt{n}) + \Theta(1)$

   (d) $T(n, m) = T(\frac{n}{2}, m) + \Theta(m)$

   (e) $T(n) = (\sqrt{n}+1)T(\sqrt{n}) + \sqrt{n}$

   Solution:

   (a) Using recursion tree, $T(n) = c + 2c + 4c + \ldots + 2^{n-1}c = (2^n - 1)c = \Theta(2^n)$.

   (b) Using recursion tree, $T(n) = \text{height} \cdot c = \log\log n \cdot c = \Theta(\log\log n)$.
   Alternatively, substitute $n = 2^m$, then we have $T(2^m) = T(2^{m/2}) + \Theta(1)$. Write $S(m) = T(2^m)$, we have $S(m) = S(m/2) + \Theta(1)$. By Case 1 of Master Theorem, we have $S(m) = \Theta(\log m)$. Hence, $T(n) = S(\log n) = \Theta(\log\log n)$.

   (c) Using recursion tree, $T(n) = c+2c+4c+\ldots+2^{\log\log n}c = (2^{\log\log n - 1} - 1)\cdot c = \Theta(2^{\log\log n}) = \Theta(\log n)$.
   Alternatively, substitute $n = 2^m$, then we have $T(2^m) = 2T(2^{m/2}) + \Theta(1)$. Write $S(m) = T(2^m)$, we have $S(m) = 2S(m/2) + \Theta(1)$. By Case 1 of Master Theorem, we have $S(m) = \Theta(m)$. Hence, $T(n) = S(\log n) = \Theta(\log n)$.

   (d) Using recursion tree, $T(n) = \text{height} \cdot m = \Theta(m \log n)$.

   (e) We guess $T(n) = \Theta(n)$ and prove it by substitution method.

   Lower bound: We want to show that $T(n) \geq cn$.

   $$\begin{aligned} T(n) &= (\sqrt{n}+1)T(\sqrt{n}) + \sqrt{n} \\ &\geq (\sqrt{n}+1)(c\sqrt{n}) + \sqrt{n} \\ &\geq cn + c\sqrt{n} + \sqrt{n} \\ &\geq cn \end{aligned}$$

   Upper bound: We want to show that $T(n) \leq c_1 n - c_2$.

   $$\begin{aligned} T(n) &= (\sqrt{n}+1)T(\sqrt{n}) + \sqrt{n} \\ &\leq (\sqrt{n}+1)(c_1\sqrt{n} - c_2) + \sqrt{n} \\ &\leq c_1 n + (c_1 - c_2 + 1)\sqrt{n} - c_2 \\ &\leq c_1 n - c_2 \qquad \blacktriangleleft \text{ choose } c_2 \geq c_1 + 1 \end{aligned}$$

4. **Recurrences** (AY 19/20 Sem 2). Solve the following recurrence relations. Provide as tight a bound as possible. If you use the master theorem, state the case that applies, along with a short reasoning why the case applies. Otherwise, give a detailed proof (using any of the methods). Unless otherwise specified, you can assume base cases, $T(n)$ for $n \leq$ some constant, to be $\Theta(1)$.

   (a) $T(n) = 9T(n/3) + n^2/\log n$.

   (b) $T(2^n) = T(2^0) + T(2^1) + \cdots + T(2^{n-1}) + n^2$.

   (c) $T(n) = 5T(n/4) + n^4/\log\log n$.

   (d) $T(n) = 4T(n/4) + n\log\log\log n$.

   Solution:

(a) Divide both sides by n^2 and apply telescoping. We have

$$\begin{aligned}\frac{T(n)}{n^2} &= \frac{1}{\log n} + \frac{1}{\log(n/3)} + \frac{1}{\log(n/9)} + \cdots \\ &= \log_3 2 \cdot \left(\frac{1}{\log_3 n} + \frac{1}{\log_3 n - 1} + \frac{1}{\log_3 n - 2} + \cdots \right) \\ &= \Theta(\log\log n) \qquad \blacktriangleleft \text{ harmonic series}\end{aligned}$$

Hence $T(n) = \Theta(n^2 \log\log n)$.

(b) Notice $T(2^n) - T(2^{n-1}) = T(2^{n-1}) + n^2 - (n-1)^2$.
Hence, $T(2^n) = 2T(2^{n-1}) + 2n - 1$.
Substitute $N = 2^n$, we get $T(N) = 2T(N/2) + 2\log N - 1$. Using master theorem, case 1: $2\log N - 1 = O(N^{\log_2 2} - \varepsilon)$ for $\varepsilon = 0.5$, so $T(N) = \Theta(N)$.
Therefore, $T(2^n) = \Theta(2^n)$.

(c) Using Master Theorem, Case 3: $f(n) = n^4/\log\log n = \Omega(n^{\log_4 5+\varepsilon})$ for $\varepsilon = 1$.
Furthermore, the regularity condition is satisfied: $af(n/b) = 5(n/4)^4/\log\log(n/4) \leq \frac{5}{256} \cdot \frac{n^4}{\log\log n} < cf(n)$ for $c = \frac{6}{256}$.
Therefore, $T(n) = \Theta(n^4/\log\log n)$.

(d) Divide both sides by n and apply telescoping. We have

$$\begin{aligned}\frac{T(n)}{n} &= \log\log\log n + \log\log\log(n/4) + \log\log\log(n/16) + \cdots \\ &= \Theta(\log n \log\log\log n) \qquad \blacktriangleleft \text{ by a similar argument in tutorial}\end{aligned}$$

Hence, $T(n) = \Theta(n \log n \log\log\log n)$.

5. **Proof of Correctness** (AY 23/24 Sem 2 Practice Set). Consider the function Fun(x, y). What is its output? What is the invariant for the while loop? Can you show that this algorithm correctly compute the output?

Algorithm 1

```
1: procedure FUN(x, y)
2:     ans ← 0, p ← x, q ← y
3:     while q > 0 do
4:         r ← q mod 2
5:         q ← q/2
6:         ans ← ans + r × p
7:         p ← p × 2
8:     return ans
```

Solution: The output of the function is $x \times y$.

Let $b_k b_{k-1} \ldots b_2 b_1 b_0$ be the binary representation of y. Then, the binary representation of $y/2^i$ is $b_k \ldots b_i$ and the binary representation of $y \mod 2^i$ is $b_{i-1} \ldots b_1 b_0$.

The invariant is: For the i-th iteration, q is $y/2^i$, p is $x \times 2^i$, and ans is $x \times (y \mod 2^i)$.

Initiation: In the 0-th iteration, $i = 0$. We have q is y, p is x and ans is $x \times (y \mod 2^0) = 0$. This is exactly the assignment in step 1.

Maintenance: After the i-th iteration, q is $y/2^i$, p is $x \times 2^i$, and ans is $x \times (y \mod 2^i)$. Then, the $(i+1)$-th iteration goes through steps 3-5. After step 3, r is $q \mod 2 = (y/2^i) \mod 2 = b_i$. q is set to be $q/2 = (y/2^i)/2$, which is $y/2^{i+1}$.

After step 4, $ans = x \times (y \mod 2^i) + b_i \times x \times 2^i = x \times (y \mod 2^i + b_i 2^i) = x \times (y \mod 2^{i+1})$.

After step 5, p is set to be $p \times 2 = x \times 2^{i+1}$.

In summary, after the $(i+1)$-th iteration, q is $y/2^{i+1}$, p is $x \times 2^{i+1}$, and ans is $x \times (y \mod 2^{i+1})$.

Termination: The while loop terminates when $q = 0$ which is the $(k+1)$-th iteration. According to the invariant, ans is $x \times (y \mod 2^{k+1}) = x \times y$. The algorithm computes the correct output.

6. **Divide and Conquer, Proof of Correctness** (AY 24/25 Sem 2, Modified).
You are consulting for the NUS Office of Student Affairs (OSA), which is investigating cheating cases involving bootleg, defaced student cards. The OSA has confiscated a collection of n student cards suspected of being unauthorized duplicates. Each student card is a small plastic card embedded with a chip that securely stores encrypted data. Each card's encrypted data correspond to a unique student (student number), and multiple student cards may correspond to the same student number. Since the cards are defaced, it is difficult to read the student number directly from the cards. Instead, OSA uses a 'matching device', a device capable of determining whether two student cards correspond to the same student number after performing some computations.
The OSA poses the following question: among the collection of n confiscated cards, is there a subset of more than $n/2$ cards that all correspond to the same student number?
Due to security concerns, you are not able to know the student number on a card, but are only allowed to interact with the matching device. Specifically, the only operation you can perform is to select two student cards and use the matching device to determine whether they correspond to the same student number.

   (a) Design an efficient method to aid OSA's investigation using $O(n \log n)$ queries to the matching device. Analyze the query complexity of your algorithm (i.e., the number of queries made by your algorithm in terms of n).

   (b) Give a formal proof of correctness on why your algorithm is correct.

Solution 1 (Divide and Conquer):

Suppose the function match(X, Y) returns true iff the student numbers on cards X and Y are the same. Furthermore, suppose the n student cards are in $A[1..n]$. The following algorithm returns X such that if there exists a student number corresponding to more than half of the cards in $A[i \dots j]$, then X is a card corresponding to such a student number (otherwise, the card may be arbitrary).

Consider the following algorithm:

```
Majority(A, i, j)
1. If i = j, then return A[i]
2. mid = floor((i + j) / 2)
3. maj1 = Majority(A, i, mid)
4. maj2 = Majority(A, mid + 1, j)
5. count1 = the number of occurences of maj1 in A[i...j]
6. count2 = the number of occurences of maj2 in A[i...j]
7. If count1 > count2, output maj1, else output maj2.
```

If there is a student number that corresponds to the majority of cards in $A[i \dots j]$, then the above algorithm returns a student card corresponding to this student number; otherwise, it may return an arbitrary student card.

Proof of Correctness: We now show inductively on the difference $j - i$ that the algorithm works correctly. If $j - i = 0$, then step 1 gives the answer correctly. Now inductively consider the algorithm based on the difference $j - i$, where we assume that the algorithm returns correctly when this difference is smaller. By induction, the two calls in steps 3 and 4 returned correct answers to the smaller portions of the array. If there is a student number which corresponds to a majority of the cards in $A[i \dots j]$, then it must also correspond to a majority of the cards in at least one of $A[i \dots mid]$ and $A[mid + 1 \dots j]$ (*Note: Prove this claim by a contradiction*). Thus, steps 5 to 7 will correctly find a card corresponding to such a student number, if any.

Query Complexity: Let $T(m)$ denote the query complexity of the algorithm when $j - i + 1 = m$. We have $T(n) = 2T(n/2) + \Theta(n)$. Using Case 2 of Master Theorem, $T(n) = \Theta(n \log n)$.

Solution 2 (Iterative):

Consider the following algorithm:

```
Majority(A):
   maj = 1
   count = 1
   for i from 2 to n,
```

```
    if A[i] == A[maj],
        count = count + 1
    else,
        count = count - 1
        if count == 0,
            maj = i
            count = 1
return A[maj]
```

Proof of Correctness: We claim the following invariant for the for loop: After the i-th iteration, the cards $A[1 \dots i]$ can be partitioned into `count` cards equal to `A[maj]` and $\frac{\texttt{i} - \texttt{count}}{2}$ pairs of unequal cards.

- **Initialization**: Before the for loop starts, the card $A[1]$ can be partitioned into one card equal to `A[maj]` $=$ `A[1]`. So the invariant trivially holds for `i` $= 1$.
- **Maintenance**: We assume that there exists a partitioning for the `i`-th iteration. Consider the $(\texttt{i} + 1)$-th iteration.
  If `A[i+1] == A[maj]`, then we could simply insert one more card equal to `A[maj]`.
  Otherwise, we pair `A[i]` with a card that is equal to `A[maj]`. The number of pairs of cards we get is $\frac{\texttt{i} - \texttt{count}}{2} + 1 = \frac{(\texttt{i} + 1) - (\texttt{count} - 1)}{2}$, and the number of individual cards equal to `A[maj]` is `count` $- 1$. Therefore, there is a valid partitioning after we decrement `count` by 1.
- **Termination**: Suppose there exists a majority element that is not `A[maj]`. Consider the partitioning at the end of the n-th iteration. The element must only appear once in each pair of unequal cards, so it appears at most $\frac{n - \texttt{count}}{2} \leq \frac{n}{2}$ times, a contradiction.
  Therefore, if a majority element exists, then it must be `maj`.

Query Complexity: There are $n - 1$ iterations in the loop and each iteration makes one query. Hence, the query complexity is $\Theta(n)$.

7. **Divide and Conquer** (AY 23/24 Sem 2 Practice Set). Given an array A of n integers (possibly 0 or negative as well), find the largest possible value c that can be obtained by summing up the values in some contiguous subarray of A, i.e., $c = A[i] + A[i+1] + A[i+2] + \cdots + A[i+t]$ for some i, t. Think of a divide and conquer solution that does it in $O(n \log n)$ time. As a bonus, you can then think about how to improve it to $O(n)$ by some minor modifications.

Solution: We can divide the array into a left sub array and right sub array from the center. If we keep dividing the array when the length is one the solution is trivial, the maximum sum is equal the the element in the array. Now when we combine, the maximum possible contiguous subarray could be from the right sub array or the left sub array. It can also be a contiguous sub array spanning across the center point. Therefore in the combining stage we need to consider all three possibilities. To find the largest possible contiguous sub array across the center we can scan to the left and right of the center and calculate the maximum sum in linear time.

```
MaxSubArray(arr, left, right)
    left_sum = MaxSubArray(arr, left, mid)
    right_sum = MaxSubArray(arr, mid + 1, right)
    mid_sum = FindMidSum(arr, left, right)
    return max(left_sum, right_sum, mid_sum)

FindMidSum(arr, left, right)
    max_r = -inf, sum = 0
    for i from mid + 1 to right
        sum += arr[i]
        if sum > max_r, max_r = sum
    max_l = -inf, sum = 0
    for i from mid downto left
        sum += arr[i]
        if sum > max_l, max_l = sum
```

```
    return max_l + max_r
```

The recurrence is $T(n) = 2T(n/2) + \Theta(n)$. By case 2 of Master Theorem, $T(n) = \Theta(n \log n)$.

Bonus: Notice that the function `FindMidSum` aims to find the maximum sum prefix of the first half and the maximum sum suffix of the second half. We can speed up the combine step to $\Theta(1)$ by delegating this step to the recursion. Now, the `MaxSubArray` function should return 4 values – the maximum sum subarray, the maximum sum **prefix**, the maximum sum **suffix** and the maximum sum **prefix-suffix** (i.e. the sum of the entire range). The combine step becomes as follows:

- `this.maxSubarray = max(left.maxSubarray, right.maxSubarray, left.maxSuffix + right.maxPrefix)`
- `this.maxPrefix = max(left.maxPrefix, left.whole + right.maxPrefix)`
- `this.maxSuffix = max(right.maxSuffix, right.whole + left.maxSuffix)`
- `this.whole = left.whole + right.whole`

The recurrence becomes $T(n) = 2T(n/2) + \Theta(1)$. By case 1 of Master Theorem, $T(n) = \Theta(n)$.

8. **Lower bounds** (AY 23/24 Sem 2 Practice Set, Modified). Consider an array of distinct integers sorted in increasing order. The array has then been rotated (anti-clockwise) k number of times, i.e., all the numbers in the sorted array have been (cyclically) shifted k places on the leftside. Now given such an array, find the value of k. Prove that your algorithm is optimal (asymptotically).

Solution Sketch: At the start the array is sorted. Therefore the largest element will be at the end of the array. When the array is rotated anti-clockwise once, the largest element will move one step to the left. If the array is rotated again, the largest element will take another step to the left and so on. Therefore, if we find the position of the largest element (i_{max}) we can calculate k as $k = n - i_{max}$. We can use binary search for an $O(\log n)$ solution.

There are at least n leaf nodes in the decision tree ($k = 0, 1, \ldots, n-1$), so the lower bound for any comparison-based algorithm is $\Omega(\log n)$.

9. **Lower bounds** (New Question). There are N students $\{1, 2, \ldots, N\}$ interning at N companies $\{1, 2, \ldots, N\}$. You know that each student interns at exactly one company, and no two students intern at the same company. Each student knows which company he/she is ownself interning at, but there is no information about the other students. For each round, you may ask any student a Yes/No question about the company the student interns at.

   (a) Design an algorithm that, with as few Yes/No questions as possible, finds out the mapping between the students and the companies. Your algorithm only has to be asymptotically optimal (multiplicative and constant factors do not matter).

   (b) Prove that your algorithm is optimal asymptotically, i.e. any correct algorithm must take, asymptotically, at least as many Yes/No questions.

Solution Sketch: There are $N!$ mappings between the students and the companies. There are at least $N!$ leaf nodes in the decision tree, so the lower bound for any algorithm is $\Omega(\log N!) = \Omega(N \log N)$ (by Stirling's approximation).

It is easy to design an asymptotically optimal algorithm – for each student, we ask $\lceil \log N \rceil$ questions in the form of "Do you intern at one of the companies in $\{l, l+1, \ldots, r\}$", where we reduce the search range by half using each question. This is effectively a binary search. Therefore, we ask $N\lceil \log N \rceil = \Theta(N \log N)$ questions in total.