Problem Set 2

Name: Thomas Ren

Collaborators: None

Before beginning the recurrence problem, I will write down the **Master Theorem**.
Master Theorem: Given a equation like the following form

$$T(n) = aT(\frac{n}{b}) + f(n), \quad a \geq 1, b \geq 1$$

where **a** is the branching factor, **b** is the time reduced by each recurrence, $f(n)$ is the complexity of operation needed for each node.
The complexity can be deduced as the following three situations depending on the form of $f(n)$.

1. $f(n) = O(n^{\log_b a - \epsilon}), \quad \epsilon > 0$
   $\Rightarrow T(n) = \Theta(n^{\log_b a})$

2. $f(n) = \Theta(n^{log_b a} log^k n), \quad k \geq 0$
   $\Rightarrow T(n) = \Theta(n^{\log_b a} log^{k+1} n)$, this also works by substitute Θ with $O \quad or \quad \Omega$

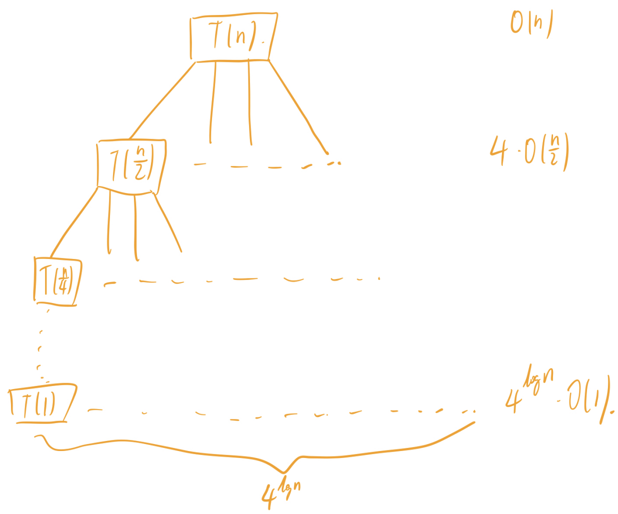
3. $f(n) = \Omega(n^{log_b a + \epsilon}), \quad \epsilon > 0 \quad and \quad af(\frac{n}{b}) < cf(n), \quad c \in (0, 1)$
   $\Rightarrow T(n) = \Theta(f(n))$

Problem 2-1.

(a) Master Theorem:
For a template of solving this sort of problems, it includes the following steps:

1. Get the parameters $a, b \ and \ f(n)$.
   Here, $a = 4, b = 2, f(n) = O(n)$
2. Decide which case does $f(n)$ fit.
   Here, $\log_b a = \log_2 4 = 2$
   $f(n) = O(n) = O(n^{2-\epsilon}), \ \epsilon > 0$
   So, **case 1.**
3. Derive $T(n)$ according to the theorem.
   $\Rightarrow T(n) = \Theta(n^{log_b a}) = \Theta(n^2)$

Recursion Tree:



Assuming $O(n) = cn$, for some constant c.
Then we can convert this recursion tree to a **geometry series**.

$$
\begin{aligned}
T(n) &= O(n) + 4O(\frac{n}{2}) + \cdots + 4^{\log n}O(1) \\
&= cn + 2cn + \cdots + 2^{\log n}cn \\
&= \sum_{i=0}^{\log n} 2^i cn \\
&= cn(2^{\log n + 1} - 1) \\
&< 2cn^2 \\
&= O(n^2)
\end{aligned} \tag{1}
$$

(b) Master Theorem:

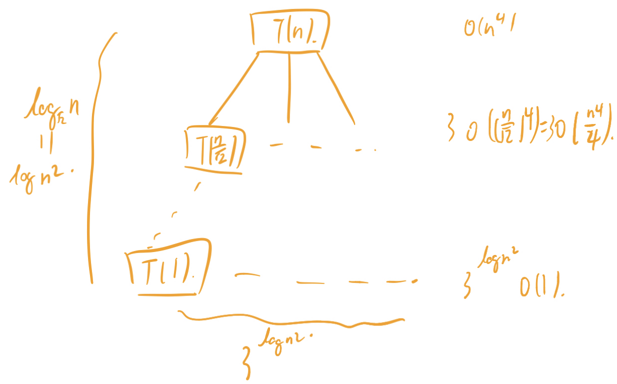
For this part, we need to upper bound T(n) and lower bound it by choosing $f(n)$ to be $\Theta(n^4)$ *or* 0.

- $f(n) = \Theta(n^4)$ to upper bound $T(n)$.
  - $a = 3, b = \sqrt{2}, f(n) = \Theta(n^4)$
  - $\log_b a = log_{\sqrt{2}}3 = \log_2 9 \in (3, 4)$
    $f(n) = \Theta(n^4) = \Omega(n^{\log_2 9})$
    $af(\frac{n}{b}) = 3f(\frac{n}{\sqrt{2}}) = \frac{3}{4}n < cn, c \in (0, 1)$
    Case 3.
  - $\Rightarrow T(n) = \Theta(f(n)) = \Theta(n^4)$
    Since this is upper bounding $T(n)$, $T(n) = O(n^4)$
- $f(n) = 0$ to lower bound $T(n)$
  - $a = 3, b = \sqrt{2}, f(n) = 0$

- $\log_b a = log_{\sqrt{2}}3 = \log_2 9 \in (3, 4)$
  $f(n) = 0 = O(\log_b a - \epsilon), \quad \epsilon > 0$
  Case 1.
- $\Rightarrow T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 9})$
  Since this is lower bounding $T(n)$, $T(n) = \Omega(n^{\log_2 9})$

- As a conclusion, $T(n) = O(n^4)$ *and* $T(n) = \Omega(n^{\log_2 9})$

Recursion Tree:



1. $f(n) = \Theta(n^4)$

$$T(n) = \Theta(n^4) + 3\Theta((\frac{n}{\sqrt{2}})^4) + \cdots + 3^{\log n^2}\Theta(1)$$
$$= cn^4 + \frac{3}{4}cn^4 + \cdots + (\frac{3}{4})^{\log n^2} cn^4$$
$$= \sum_{i=0}^{\log n^2} c(\frac{3}{4})^i n^4$$
$$< \sum_{i=0}^{\infty} c(\frac{3}{4})^i n^4$$
$$= 4cn^4 = O(n^4)$$

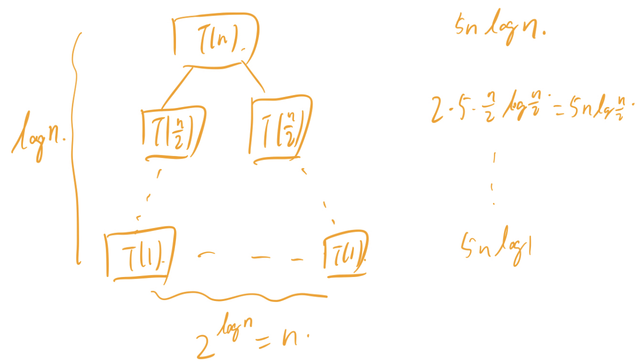
1. $f(n) = 0$
In this situation, we only need to compute the number of leaves, where each leaf is $\Theta(1)$, so $T(n) = \Omega(3^{\log n^2})$
As a conclusion, $T(n) = O(n^4)$ *and* $T(n) = \Omega(n^{\log_2 9})$

(c) Master Theorem:

1. $a = 2, b = 2, f(n) = n \log n$
2. $\log_b a = log_2 2 = 1$
   $f(n) = \Theta(n \log n) = \Theta(n^{\log_b a} \log^k n)$
   Case 2.

3. $\Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n) = \Theta(n \log n^2)$

Recursion Tree:



$$
\begin{aligned}
T(n) &= 5n \log n + 2 \times 5 \frac{n}{2} \log \frac{n}{2} + \cdots + 2^{\log n} \\
&= \sum_{i=0}^{\log n} 5n(\log n - i) \\
&= 5n(\log^2 n - \sum_{i=0}^{\log n} i) \\
&= \Theta(5n \log^2 n)
\end{aligned} \tag{2}
$$

(d) Substitution:

The template of substitution method is: First, guess the form of the solution. Second, use induction to prove that.

1. **Guess:** $T(n) = \Theta(n^2)$
2.

$$
\begin{aligned}
T(n) = T(n-2) + \Theta(n) = \Theta((n-2)^2) + \Theta(n) \\
\Rightarrow T(n) = c(n-2)^2 + dn
\end{aligned}
$$

, for some constant c and d.

$$\Rightarrow T(n) = cn^2 - 4cn + 4c + dn = \Theta(n^2)$$

Q.E.D

Problem 2-2.

(a) In this part, the $get\_at()$ operation takes constant time but $set\_at()$ operation takes $\Theta(n \log n)$ time. Thus making the $set\_at()$ operation a more important factor when considering the which sorting algorithm to use.

1. First, the merge sort algorithm is not **in-place**, so we have to rule it out.
2. Then we consider the use of set_at operations for $selection\ sort$ and $insertion\ sort$.
   Selection Sort: $O(n)$ set_at operations for swapping the largest value and the last item of subarray.
   Insertion Sort: $O(n^2)$ set_at operations for insert the minimum value in to the subarray and move the rest on the right for one.
3. Thereby, we choose **Selection Sort** in this situation.

(b) Since we have pointers here, we can set and get the items in constant time, Which makes us focus on the comparison cost which is $\Theta(\log n)$ time.

1. For both **Selection Sort and Insertion Sort**, the comparison can be $\sum_{i=1}^{n-1} i = O(n^2)$, for comparing every item with items not in the subarray.
2. **Merge Sort** takes $O(n \log n)$ comparisons for being the lower bound of comparison model in a binary tree.
3. Thereby, we choose **Merge Sort** in this situation.

(c) This part of problem stands on whether the running time of algorithms depends on the order of the input list.

1. For **Selection Sort** and **Insertion Sort**, they both need to run $\Theta(n^2)$ and $\Theta(n \log n)$ time that are independent from the input order.
   So, the running time of **Selection Sort** and **Insertion Sort** will be $\Theta(n^2)$ and $\Theta(n \log n)$.
2. For **Insertion Sort**, the running time of insertion depends on the order of input.
   For instance, if the insert index is k spaces away, then the $insert$ operation takes $\Theta(k)$ time.
   As we known from the problem, $k = O(\log \log n)$, which makes the **whole running time** to be $\Theta(n + k) = \Theta(n + \log \log n) = \Theta(n)$.
3. Thereby, we choose **Insertion Sort** in this situation.

Problem 2-3. Before dealing with this problem, I would like to introduce a search algorithm that would quickly determine the range of a certain item.
Exponential Search: for each step,say i_{th} step, we jump to the 2^i_{th} index if it's in the range, otherwise to the end of the list.
This searching algorithm will take $\Theta(\log n)$ running time to determine the range and can be easily proven by taking logarithmic on both sides.

1. **Find the range:** We will do this search starting from both ends, so this only takes $\Theta(\log k)$ running time for finding the range.
2. **Pinpoint Datum:** Pinpoint Datum using **Bisection Search**. This also takes $\Theta(\log k)$ running time.
   Since the range will be $2^{j+1} - 2^j$, so Bisection Search takes $\Theta(\log 2^{j+1} - 2^j)$ running time, which is $\Theta(1 + j),\ j = \lfloor \log k \rfloor$. That gives to $\Theta(\log k)$

Problem 2-4. Before defining the structure of database, we need to analyze the problem and the given operations.

1. We need a data structure to store the viewers. And according to the $send(v, m)$ operation and $ban(v)$ operation, we need to find the viewers in $O(\log n)$ time. **So, the data structure should be a sorted list.**
2. The $build(V)$ operation allows more than linear complexity, so we can use **merge sort** to sort the list, which is $O(n \log n)$, instead of randomly append the a new viewer to the list. This means for initial viewers, we append them to a list and use merge sort, then for every new viewers, we use insertion sort.
   The running time should be $O(n \log n + n) = O(n \log n)$.
3. Then we need another data structure to store the chatting. Since the $ban(v)$ operation needs to delete all the messages sent by v, the data structure to store messages has to be **link list**.
4. Besides, to reach the messages in $O(1)$ time, we have to adhere a pointer to the viewers message when using $send(v, m)$ operation.

Here, we give the implemented data structure as follows:

1. A sorted array S sorted by user's key, think of user's ID as a machine word. Each item in the list is a tuple of viewer's ID and pointer pair, namely, (v, p_v). This pointer points to a link list, which can be either singly linked or doubly linked, and this link list stores all the messages sent by this viewer.
2. This link list L_v is the link list that stores the all the messages sent by a certain viewer. And each item in the link list also has pointer pointing to the same item in a larger link list, note that this larger link list L has to be doubly linked, that stores all the messages sent by any viewers in the room.
   For example, assuming L_v is a singly linked list, each node has attribute $self.next$ is the next message; $self.item$ is the current message; $self.chro$ is the node in the larger link list L.
3. The larger link list L contains all the messages, so that $recent(k)$ operation can be easily satisfied by this data structure.

Problem 2-5.

(a) Inspired by linear sorting and the **Problem 5** in problem session 3, we can build a constant time frequency table which have length of the **largest terminate time to least start time**.
Then, we can easily implement a linear time algorithm to get the booking schedule B.
To be more specifically, this algorithm get the booking schedule B in $O(cn)$ time, where $c = \Omega(t_{max} - s_{min})$.
Before implementing this part, we have first implemented the second part, which is the $satisfying\_booking(R)$ algorithm in $O(n)$ time.
Then we can break the $B1$ and $B2$ to two different **frequency tables** and simply concatenate these two frequency tables in **linear time**.
Good news is that the merged two frequency tables serves as the same function as a set of talk requests.
At last, we can use this set of talk requests to get the booking schedule $B = B1\, merge\, B2$ in **linear time**.

(b) This part is the essence of our algorithm.
We will break this task by the following steps:

1. Build a frequency table, that count how much bookings are there in one time span in **linear time**.
2. Traverse the frequency table to get a room booking, i.e. find the discontinuous indices in the frequency table in **constant time**.
3. Each chunk of discontinuous indices should be appended with start time and terminate time to the final list.
4. Turn the list to tuple from list in **linear time**.

Thereby, we finished our solution.

(c) Our code are as follows and has passed the unittest.

```python
def satisfying_booking(R):
    '''
    Input:  R | Tuple of |R| talk request tuples (s, t)
    Output: B | Tuple of room booking triples (k, s, t)
              | that is the booking schedule that satisfies R
    '''
    B = []
    ##################
    # YOUR CODE HERE #
    ##################
    start_time = []
    terminate_time = []
    minimum_start_time = 0
    maximum_terminate_time = 0
```

```
    for (start,terminate) in R:              # O(n)
        start_time.append(start)
        terminate_time.append(terminate)
        if terminate>=maximum_terminate_time:
            maximum_terminate_time = terminate
        if start<=minimum_start_time:
            minimum_start_time = start

    time_span = maximum_terminate_time-minimum_start_time # O(1)
    booking_agenda = [0]*time_span

    for (start, terminate) in R:             # O(cn), for some constant c
        for i in range(start,terminate):     # O(1)
            booking_agenda[i] += 1

    start = minimum_start_time
    terminate = minimum_start_time + 1
    relative_time = 0
    i = 0

    while terminate<maximum_terminate_time: # O(1)
        if booking_agenda[start+relative_time] == booking_agenda[start+relative_time+1
            terminate += 1
            relative_time += 1
        else:
            relative_time = 0
            if booking_agenda[start+relative_time] != 0:
                B.append((booking_agenda[start+relative_time],start,terminate))
            start = terminate
            terminate += 1
    B.append((booking_agenda[start+relative_time],start,terminate))

    return tuple(B)

def merge_booking(B1,B2):
    '''
    :param B1: a booking schedule
    :param B2: a booking schedule
    :return: a booking schedule that merges the B1 and B2
    '''
    R1 = []
    R2 = []
    for num, start, terminate in B1:
        while num > 0:
            R1.append((start,terminate))
            num -= 1
    for num, start, terminate in B2:
        while num > 0:
            R2.append((start,terminate))
            num -= 1
```

```
    R = R1 + R2 # O(n)
    B = satisfying_booking(R)

    return B
```

Submit your implementation to `alg.mit.edu`.