Problem Set 2

Name: Gabriel Chiong

Collaborators: None

Problem 2-1.

(a) Using the recursion tree method, the tree has depth $\log_2 n$, with $4^i \cdot f(\frac{n}{2^i})$ work being done at each level. Therefore, the total work done is $\sum_{i=0}^{\log_2 n} 4^i \cdot c\frac{n}{2^i} = cn\left(\frac{1-2^{log_2 n+1}}{1-2}\right)$ by the identity for a geometric series. This simplifies to $-cn(1-(n+1)) = cn^2 = O(n^2)$.
Using the Master Theorem, we have $\log_b a = \log_2 4 = 2$, and $f(n) = n^{\log_2 4-1}$ where $\varepsilon = 1$. Therefore, we are in Case 1 and $T(n) = O(n^2)$.

(b) Using the recursion tree method, the tree has depth $\log_{\sqrt{2}} n = 2\log_2 n$. At each level, the work done is $3^i \cdot c\left(\frac{n}{2^{i/2}}\right)^4 = cn^4 \cdot \left(\frac{3}{4}\right)^i$. Therefore, the total work done is $\sum_{i=0}^{2\log_2 n} cn^4 \cdot \left(\frac{3}{4}\right)^i < cn^4 \sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i = \frac{cn^4}{1-3/4} = 4cn^4 = O(n^4)$.
Using the Master Theorem, we have $\log_b a = \log_{\sqrt{2}} 3 = 2\log_2 3$. This is less than 4. Therefore, we have $f(n) = n^4 = n^{\log_{\sqrt{2}} 3+\varepsilon}$, where $\varepsilon = 4 - \log_{\sqrt{2}} 3$. We are in Case 3 and $T(n) = O(n^4)$.

(c) Using the recursion tree method, the tree has depth $\log_2 n$. At each level, the work done is $2^i \cdot 5\left(\frac{n}{2^i}\right)\log\left(\frac{n}{2^i}\right)$. Therefore, the total work done is $\sum_{i=0}^{\log_2 n} 2^i \cdot \frac{5n}{2^i}(\log n - \log 2^i) = 5n\left(\sum_{i=0}^{\log_2 n}\log n - \sum_{i=0}^{\log_2 n} i\right) = 5n\left(\log^2 n - \frac{(\log n+1)\log n}{2}\right)$, by the summation identity. This can be eventually simplified to $\frac{5n}{2}\log^2 n - \frac{5n}{2}\log n = \Theta(n\log^2 n)$.
Using the Master Theorem, we have $\log_b a = \log_2 2 = 1$, and $f(n) = 5n\log n$ which matches Case 2: $f(n) = \Theta(5n^{\log_2 2}\log^k n) = \Theta(n\log^2 n)$. By choosing $k = 1$, we obtain the solution $T(n) = \Theta(n\log^2 n)$.

(d) We guess that the solution to the recurrence is cn^2. We then rearrange the equation

$$\begin{aligned} cn^2 &= c(n-2)^2 + \Theta(n) \\ cn^2 &= cn^2 - 4cn + 4c + \Theta(n) \\ \Theta(n) &= 4cn - 4c \end{aligned}$$

which is true, therefore we know that the solution is $\Theta(n^2)$.

Problem 2-2.

(a) An in-place sorting algorithm rules out using merge sort. Using insertion sort requires $O(n^2)$ number of `get_at(i)` operations, and $O(n^2)$ number of `set_at(i, x)` operations for repeated swaps. Therefore, the overall runtime of using insertion sort is $O(n^3 \log n$ in the worst-case.

Using selection sort results in $O(n^2)$ number of `get_at(i)` operations, and $O(n)$ number of `set_at(i, x)` number of operations, since we swap once per iteration. This results in an overall runtime of $O(n^2 \log n)$ in the worst-case using insertion sort.

Therefore, we choose selection sort in this case.

(b) There is no requirement for in-place sorting, so merge sort can be used with a worst-case $O(n \log n)$ number of comparisons. Both insertion and selection sort require $O(n^2)$ number of comparisons in the worst-case, which has a poorer performance than merge sort.

(c) Both selection and merge sort take $\Theta(n^2)$ and $\Theta(n \log n)$, respectively in the worst-case - regardless of inputs.

However, insertion sort can break its loop early, so it could take $O(n)$ time for certain inputs. For k inverted pairs (or pairs that are out of order), insertion sort takes $O(n+k)$ time. Since $k = \log \log n$, insertion sort runs in $O(n + \log \log n) = O(n)$ time. Therefore, we choose insertion sort in this case.

Problem 2-3. One possible solution is to search in exponential steps, alternating between 2^i and $n - 2^i$ for $i = 1, 2, \ldots, n-2, n-1$. Once the device indicates that the 2^j-th step is too far south or the $n - 2^j$-th step is too far north, we have identified that either $2^{j-1} < k < 2^j$ or $n - 2^j < k < n - 2^{j-1}$. Identifying the correct interval takes $O(j)$ time in the worst-case.

Applying binary search to the remaining interval of 2^{j-1} takes $O(\log 2^{j-1}) = O(j)$ time in the worst-case. Since we have $2^{j-1} < k < 2^j$ (or equivalently, $n - 2^j < k < n - 2^{j-1}$), then taking the $\log$ of the statement yields: $j - 1 < \log k < j$, which implies $j = O(\log k)$ as desired.

The algorithm is correct since the first step will identify the bounded range of Datum, and the second step is correct by using binary search on the correctly identified intervals.

Problem 2-4. The database can be implemented using a doubly-linked list L of messages sorted in chronological order, with the most recent ones at the head of the list, as well as a static array S which contain tuples of a user's id, and a singly-linked list of pointers to the user's messages in L.

To implement `build(V)`, initialize L to an empty list in $O(1)$ time, and S to contain all $n = |V|$ users in $O(n)$ time. `None` pointers can be used to signify banned users. To maintain the sorted order of S by user id, apply merge sort in $O(n \log n)$ time. Therefore, the overall worst-case time of the `build(V)` operation is $O(n \log n)$. This is correct as it maintains the invariant of the database.

To implement `send(v, m)`, append the message to L's head in $O(1)$ time. Use binary search to identify the user in S since it is in sorted order, and append a pointer to the message in L in $O(1)$ time (the head of the user's pointer list in S). This takes $O(\log n)$ time in the worst-case, and is correct as it maintains the database invariant.

To implement `recent(k)`, simply traverse through L and return items until either the end of L is reached, or k elements have been returned. This takes $O(k)$ time in the worst case, and is correct by the database invariant.

To implement `ban(v)`, search for the viewer in S using binary search in $O(\log n)$ time. Then traverse through the user's messages list and delete each of the messages from L (and also re-linking adjacent nodes) in $O(1)$ time. If there are n_v items in the user's pointer list in S, then the total time to delete from both lists is $O(n_v)$, since each delete takes $O(1)$ time in the worst-case. Therefore, the total operation will take $O(\log n + n_v)$ time, and is correct as in maintains the database invariant.

Problem 2-5.

(a) One possible way to solve this is to define two variables, time x and an empty schedule B. At each time step x, we maintain the invariant that B is a satisfying booking for $R_1 \cup R_2$ up to time x. As long as this invariant is maintained, the algorithm will be correct. At initialization ($x = 0$), this is trivially true since time is nonnegative. If the invariant is true every time x is increased, once x increases past the latest finishing time of all requests and the algorithm terminates, B will contain a satisfying schedule for $R_1 \cup R_2$.

We maintain two pointers, i_1, i_2 which index the earliest booking under consideration in B_1, B_2 respectively, and repeatedly increase x until both B_1 and B_2 are depleted.

The algorithm can branch five ways, on each iteration of x:

- Case 1: One schedule is depleted. In this case, take the next $(k, \max(x, s), t)$ and append it to B. Set $x = t$ and increment the index of the schedule that was taken.
- Case 2: Neither schedule has been depleted.
  - Case 2.1: No booking overlaps with x. Set $x = \min(s_1, s_2)$ (the minimum start time of either booking).
  - Case 2.2: Next booking does not overlap the other after time x. Then we can take this booking (k, x, t) and append it to B, and set $x = t$. Increase the index of the schedule taken.
  - Case 2.3: Next booking overlaps with the other after time x. Append the non-overlapping part (k, s_1, s_2) to B and set $x = s_2$.
  - Case 2.4: Both bookings overlap from time x until $t^* = \min(t_1, t_2)$. Append the overlapping part $(k_1 + k_2, x, t^*)$ to B and set $x = t^*$. Then increase the schedule index that ended at t^*.

Since the invariant is maintained on each iteration of the loop, when the algorithm terminates, B is a satisfying booking for $R_1 \cup R_2$. Each branch of the loop is executed in $O(1)$ time, and x strictly increases, therefore the overall worst-case time complexity is $O(n)$.

In the final step, we loop through B and merge adjacent bookings with equal number of rooms, k in $O(n)$ time.

(b) We can use a solution similar to merge sort, with the answer to part (a) acting as the "merge" step. Recursively divide R into roughly equal halves until one element remains, then call the procedure in part (a) on pairs of subproblems to eventually merge them into the final sorted B. At the base case when $|R| = 1$, we return $(1, s, t)$ in $O(1)$ time.

The recurrence relation obtained from this algorithm is $T(n) = 2T(n/2) + O(n)$. By the Master Theorem, this falls into Case 2, therefore the algorithm runs in $O(n \log n)$ time.

(c) Submit your implementation to `alg.mit.edu`.