Please enter your name and uID below.

Name: Qianlang Chen
uID: u1172983

Submission notes

- Due at 11:59 pm on Friday, September 11.
- Solutions must be typeset using one of the template files. For each problem, your answer must fit in the space provided (e.g. not spill onto the next page) *without* space-saving tricks like font/margin/line spacing changes.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the grading process, so please retain these until an assignment has been returned.
- Please remember that for problem sets, collaboration with other students must be limited to a high-level discussion of solution strategies. If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration. You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.

1. (Big-Θ) Proof: to use the definition of Big-Θ, we need to show both $\log(n!) = O[n\log(n)]$ and $\log(n!) = \Omega[n\log(n)]$.

   - **Part I:** show that $\log(n!) = O[n\log(n)]$, that is, there exist constants $c, k > 0$ such that $\log(n!) \leq c \cdot n\log(n)$ for all $n \geq k$.
     Proof: let $c = 1$ and $k = 100$. Now, for all $n \geq 100$, we have

$$\begin{aligned} \log(n!) &\leq 1 \cdot n\log(n) \\ \log(n!) &\leq \log(n^n) \\ 10^{\log(n!)} &\leq 10^{\log(n^n)} \\ n! &\leq n^n \\ \underbrace{n \cdot (n-1) \cdot (n-2) \cdots 1}_{n \text{ items}} &\leq \underbrace{n \cdot n \cdot n \cdots n}_{n \text{ items}} \end{aligned}$$

     (Since each factor gets smaller on the left-hand side whereas it stays constant on the right-hand side.)
     Therefore, by definition of Big-Oh, $\log(n!) = O[n\log(n)]$.
   - **Part II:** show that $\log(n!) = \Omega[n\log(n)]$, that is, there exist constants $c, k > 0$ such that $\log(n!) \geq c \cdot n\log(n)$ for all $n \geq k$.
     Proof: let $c = \frac{1}{4}$ and $k = 100$. Now, for all $n \geq 100$, we have

$$\begin{aligned} \log(n!) &\geq \frac{1}{4} \cdot n\log(n) \\ \log(n!) &\geq \log(n^{\frac{n}{4}}) \\ 10^{\log(n!)} &\geq 10^{\log(n^{n/4})} \\ n! &\geq n^{\frac{n}{4}} \\ n! &\geq (\sqrt{n})^{\frac{n}{2}} \\ \underbrace{n \cdot (n-1) \cdots (\frac{n}{2}+1)}_{\frac{n}{2} \text{ items}} \cdot \frac{n}{2} \cdots 1 &\geq \underbrace{\sqrt{n} \cdot \sqrt{n} \cdots \sqrt{n}}_{\frac{n}{2} \text{ items}} \end{aligned}$$

     (Since $\frac{n}{2} > \sqrt{n}$ for all $n \geq 100$, meaning that the first $\frac{n}{2}$ factors on the left-hand side are all greater than $\sqrt{n}$.)
     Therefore, by definition of Big-Ω, $\log(n!) = \Omega[n\log(n)]$.

   Since we have managed to show both $\log(n!) = O[n\log(n)]$ and $\log(n!) = \Omega[n\log(n)]$, by definition of Big-$\Theta$, $\log(n!) = \Theta[n\log(n)]$. $\square$

2. (Party Planning) **(a)** Proof: first, this algorithm's result is guaranteed to not contain a single lonely person, because the algorithm would've uninvited that person from the `while` loop which constantly checks for lonely people.

   Moreover, this algorithm's result is guaranteed to be the *largest* correct subset. The algorithm starts with the complete set of friends and uninvites a person only if it's necessary, that is, only if that person does not have enough friends. It then stops at the first moment when everyone in the invite-list has enough friends, meaning that the subset cannot possibly have a single extra person that is not lonely because the algorithm would've halted by then.

   Combining the above two points, the algorithm is correct as it always returns the largest subset of friends in which no one is lonely. □

   (b) The basic idea is to use a Depth-First Search. Using Java's terminologies, we can set up the local variables and helper functions as follows:

   - `invited` is a `LinkedHashMap<String, Vertex>`, representing a graph of friendships with each person being a vertex and each friendship being an edge.
     - The `Vertex` class has the following fields: `bool visited, int indegree, ArrayList<Vertex> adjVertices`.
     - We assume that the input of the program gives a list of names, and the `String` here maps them to the people's representative vertices.
     - A `LinkedHashMap` allows $O(1)$-time insertion and lookup, and it's more friendly for traversals than a regular hash-map.
   - `initialize()` takes in a list of edges as the friendships. It sets up the `invited` graph so that each vertex has the friends of that person being in the `adjVertices` and the number of friends being the `indegree`. (Time complexity: $O(n)$ with n being the number of edges/friendships.)
   - Before the `while` loop, setup a `Stack<Vertex>` for the Depth-First Search. It initially contains vertices with `indegree` of four or less. (Time complexity for adding initial values: $O(n)$ since the number of vertices is at most $2n$ for this problem.)
   - Perform the DFS: `while` the stack is not empty (meaning there are still lonely people to uninvite), pop the stack and call `uninvite()` on the element.
     - `uninvite()` takes in a `Vertex`, marks it as `visited`, goes through its `adjVertices`, and for each *unvisited* adjacent vertex, decrements its `indegree`, and adds it to the stack if its `indegree` becomes four or less.
     - The time complexity of a DFS is $O(n)$ (or more precisely, $O(|V|+n)$, but the number of vertices is at most $2n$ for this problem).
   - Finally, `convert_to_set()` returns a set of the friend names whose representative vertices are not `visited` (a vertex is only marked `visited` when it's detected lonely). (Time complexity: $O(n)$.)

   Since all the sub-procedures described above run in at most $O(n)$, the total run time of the algorithm is $O(n)$ with this setup. □

3. (Amortized ArrayList) Proof: the `ArrayList` needs to resize once the number of contained elements reaches n; so does it once it reaches $(n+c)$, $(n+2c)$, and so on. During k `add_end()` calls, the `ArrayList` has to resize approximately $\frac{k-n}{c}$ times, since each resize gives it the space for c additional elements. (The error in the number of times it needs to resize due to $(k-n)$ not being a multiple of c will get smaller as k gets large.) Since the `ArrayList` needs to resize so many times when k is large, the run time of k `add_end()` function calls is dominated by the time spent resizing and copying the elements:

$$
\begin{aligned}
T(k) &\approx \overbrace{n + (n+c) + (n+2c) + \cdots}^{\sim (k-n)/c \text{ items}} \\
&\approx (\frac{k-n}{c}) \cdot n + (c + 2c + \cdots + (\frac{k-n}{c} - 1) \cdot c) \\
&\approx \frac{kn - n^2}{c} + c \cdot (1 + 2 + \cdots + (\frac{k-n}{c} - 1)) \\
&\approx \frac{kn - n^2}{c} + c \cdot \frac{1}{2}(\frac{k-n}{c} - 1)(\frac{k-n}{c}) \\
&\approx \frac{kn - n^2}{c} + \frac{c}{2}((\frac{k-n}{c})^2 - \frac{k-n}{c}) \\
&\approx \frac{kn - n^2}{c} + \frac{k^2 - 2kn + n^2}{2c} - \frac{k-n}{c} \\
&\approx \boxed{\frac{k^2 - n^2 - 2k - 2n}{2c}}
\end{aligned}
$$

Since $T(k) \geq \frac{1}{4c} \cdot k^2$ for all $k \geq 2n^2 + 4n + 4$, by definition of Big-Ω, $T(k) = \Omega(k^2)$. Therefore, $T(k)$ is not $O(k)$. $\square$

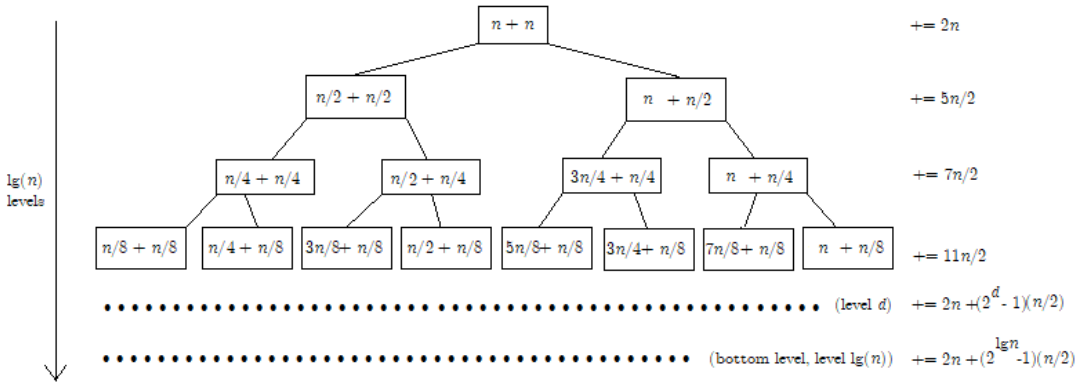
4. (Sorting) Proof: the first thing to observe is that the run time of the `sorted()` function is $O(r)$ in the worst case, where r is the value of the `right` input.

   Now, let's consider the worse-case scenario for the entire algorithm.

   `mergesort2()` is mostly the same as a regular merge-sort, and the only difference is that this new algorithm calls the `sorted()` function at each recursion level before the regular routine. In the worse-case, the `sorted()` function returns `false` every time, forcing the algorithm to do the regular merge-sort. Following this idea, let $T(n, l, r)$ represent the run time of `mergesort2()` with data size n and values of the arguments `left` and `right` being l and r, and we have

$$\begin{aligned} T(n,l,r) &= T_{\text{sorted}}(r) + T(\frac{n}{2}, l, \frac{l+r}{2}) + T(\frac{n}{2}, \frac{l+r}{2}, r) + T_{\text{merge}}(n) \\ &= O(r) + T(\frac{n}{2}, l, \frac{l+r}{2}) + T(\frac{n}{2}, \frac{l+r}{2}, r) + O(n); \\ T(n) &= T(n, 0, n) \end{aligned}$$

   This is better visualized with the following recursion tree, where the values in each node is formatted as $\boxed{T_{\text{sorted}}(r) + T_{\text{merge}}(n)}$:



   The total run time of `mergesort2()` is the sum of the run times for all levels, the values to the right of the tree:

$$\begin{aligned} T(n) &= \overbrace{2n + \frac{5n}{2} + \frac{7n}{2} + \frac{11n}{2} + \cdots}^{\log_2(n) \text{ items}} \\ &= 2n + (2n + \frac{n}{2}) + (2n + \frac{3n}{2}) + (2n + \frac{7n}{2}) + \cdots + (2n + (2^{\log_2(n)} - 1)\frac{n}{2}) \\ &= \log_2(n) \cdot 2n + (\frac{n}{2} + \frac{3n}{2} + \frac{7n}{2} + \cdots + (n-1)\frac{n}{2}) \\ &= 2n\log_2(n) + (\frac{2n}{2} - \frac{n}{2} + \frac{4n}{2} - \frac{n}{2} + \frac{8n}{2} - \frac{n}{2} + \cdots + n\frac{n}{2} - \frac{n}{2}) \\ &= 2n\log_2(n) + \frac{n}{2} \cdot (2 + 4 + 8 + \cdots + n) - \frac{n}{2} \cdot \log_2(n) \\ &= \frac{3n\log_2(n)}{2} + n(n-1) = \boxed{\frac{3n\log_2(n)}{2} + n^2 - n} \end{aligned}$$

   Now, since we have $T(n) \leq 2n^2$ for all $n \geq 1$, by definition of Big-Oh, $T(n) = O(n^2)$. $\square$

5. (Erickson 1.37) **(a)** As follows: (note: I've added an extra return value to make the algorithm work. The first two return values are as required, and `curr_depth` records the depth of the largest complete subtree rooted at the input node `curr`.)

```
Algorithm: largest_complete_subtree
Input: BinaryNode curr
Output: (BinaryNode root, int depth, int curr_depth)

if curr is null then
    return (null, -1, -1)
left_result ← largest_complete_subtree(curr.left)
right_result ← largest_complete_subtree(curr.right)
if left_result.curr_depth = right_result.curr_depth then
    new_curr_depth ← (left_result.curr_depth + 1)
else
    new_curr_depth ← (1 if curr has two non-null children; else 0)
if new_curr_depth > both children's depth then
    return (curr, new_curr_depth, new_curr_depth)
if left_result.depth > right_result.depth then
    return (left_result.root, left_result.depth, new_curr_depth)
return (right_result.root, right_result.depth, new_curr_depth)
```

(b) Proof: Let's define that, when the input is `null`, its largest complete subtree (LCS) has a `null` root and a depth of -1. This doesn't make much sense, but based on the algorithm, this implies that the LCS for a node with no children is rooted at itself with a depth of 0, which is correct for a base case.

Imagine a node N with children L and R (which may or may not be `null`). For an inductive hypothesis, let's assume that the algorithm always returns correct results when called with both L and R. That is, the algorithms correctly returns the roots (L_r, R_r) and depths (L_d, R_d) of the LCS *within* subtrees L and R, as well as the depths (L_c, R_c) of the LCS *rooted at* L and R. Now, based on these values, let's check if the algorithm produces correct results for N.

- The algorithm first checks what depth the LCS *rooted at* N can possibly have (N_c). If $L_c = R_c$, then N can join its children to make a complete subtree with depth $N_c = L_c+1$. Otherwise, N_c can only be either 1 (if N has two non-null children) or 0 (where N can only make an LCS by itself). The algorithm always produces the correct value.
- The algorithm then works out what the LCS is *within* the subtree N. It compares the depths of the LCS found by N's children (L_d and R_d) along with N_c. Out of these three candidates, the algorithm returns the LCS with the greatest depth, which is correct.

Therefore, the algorithm always returns the correct results when called with N, completing the inductive step. □

(c) The two possible worst-case inputs are (1) a tree with all nodes only having a left-child (or right-child), except for the one leaf, and (2) a completely balanced binary tree. If the input size was n, this is how the algorithm's run time would behave in each of the extreme cases:

$$\text{Case (1): } T(n) = T(n-1) + O(1);$$
$$\text{Case (2): } T(n) = 2T(n/2) + O(1)$$

Which simplifies to $T(n) = O(n)$ in both cases (algebra work omitted). □