Please enter your name and uID below.

Name: Qianlang Chen
uID: u1172983

Submission notes

- Due at 11:59 pm on Friday, October 2.
- 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. (Antimicrobial Coins BT) Let's call this algorithm MNC, which stands for the minimum number of coins. The arguments it takes in are the withdrawal W and the available coins with values $v_1, \cdots, v_n$:

$$\text{MNC}(W, v_1, \cdots, v_n) = \begin{cases} +\infty, \textbf{if } W < 0; \\ 0, \textbf{if } W = 0; \\ \min_{1 \le i \le n}\{\text{MNC}(W - v_i, v_1, \cdots, v_n)\} + 1, \textbf{if } W > 0 \end{cases}$$

The principle of designing a backtracking algorithm is to reduce the problem into a smaller but the exact same problem. This algorithm does just that: the minimum number of coins needed for a given amount is exactly one more than the minimum number of coins needed for an amount that is one coin-value away. Since there are many coin choices, meaning that there are multiple amounts being one coin-value away, the amount with this lowest MNC is the best option; and since that amount was one coin away, we add one to it to get the answer. There are several edge cases though. If the amount happens to be one of the available coin-values, the amount 0 would be one coin away, giving a result of $0 + 1 = 1$, which is handled correctly by the algorithm. Or, if the amount is more than some coin-values, the recursive call would give positive infinity, which would never be chosen to be the minimum result, since there's always a coin with value 1. Therefore, this algorithm will always produce the correct result for any positive number. $\square$

2. (Antimicrobial Coins DP, part a) As follows:

```
Algorithm: min_num_coins
  Input: int W, int V[1..n]
  Output: int

  int answers[0..W]
  answers[0] ← 0
  for int w = 1 to W do
      answers[w] ← +∞
      for each int v in V[1..n] do
          if $w \geq v$ then
              int ans ← answers[w - v] + 1
              if ans < answers[w] then
                  answers[w] ← ans
  return answers[W]
```

2. (Antimicrobial Coins DP, parts b/c)

   (b) This algorithm follows the strategy of the backtracking algorithm from the previous problem, and we made it much more efficient by using the technique of memoization. In thd DP algorithm, we use a basic array for memoizing, which is advantageous because it gives $O(1)$ run-time for both reading and writing data at any index. We want an element at index w in the array to record the minimum number of coins needed for the money amount w. Since in the original backtracking algorithm, the recursive calls only ever involve smaller values of W, we calculate the results and fill in the array from 0 to the input W. By doing it in this order, we can guarantee that the value is valid when we do a lookup, since we only ever look up from smaller indices than the current W. We also added an extra cell in the head of the array to our advantage so we wouldn't get out of bounds ever when looking up. This column also represents our base case: when an amount happens to be one of the coin-values. When the algorithm finishes, the value stored at index W in the memoizing array will have the final answer, the minimum number of coins needed for W. $\square$

   (c) This algorithm has an $O(n \cdot W)$ run-time complexity and an $O(W)$ space complexity (easy to see and prove from the code). $\square$

3. (COVID Classrooms Greedy) The algorithm is as follows. Note that the argument n is the total number of students and C is an array with the m classrooms, each having a property called `capacity`, the value of C_i.

```
Algorithm: (Greedy) min_cost
  Input: int n, classroom C[1..m]
  Output: set<classroom>

  Sort C[1..m], by capacity, in decreasing order
  set<classroom> answer
  int i ← 1
  while n > 0 do
      if i > m then
          Report error: not enough classrooms
      Add C[i] to answer
      n ← n - C[i].capacity
      i ← i + 1
  return answer
```

To prove that this greedy algorithm works, let's compare its output with some known optimal solution. Let $S_1, S_2, \cdots, S_k$ be the capacities of the classrooms from some known optimal solution for some input, sorted in decreasing order (meaning that $S_1 \geq S_2 \geq \cdots \geq S_k$). For the same input, let $G_1, G_2, \cdots, G_l$ be the capacities from our greedy algorithm's output, also sorted in decreasing order. Since we've defined $S_{1..k}$ to be optimal, we know that $k \leq l$.

Now, if $S_{1..k}$ and $G_{1..l}$ are the same, then our greedy algorithm definitely works. So, let's assume that these two outputs differ, and let i be the *first* index where $S_i \neq G_i$; in other words, $S_x = G_x$ for all $x < i$. Since our greedy algorithm has always gone for the classroom with the biggest capacity first, it can only be the case that $G_i > S_i$. This means if we replace S_i from the optimal solution with G_i, it would also do the job of providing a room to S_i students, so this replacement wouldn't make the solution S not valid. Moreover, since S is sorted in decreasing order, we know that S would not make use of any room with capacity G_i anymore ($S_x \neq G_i$ for all $x > i$), and since i is the first instance of difference, we know that there's at least one room with capacity G_i unchosen by the optimal solution. These two facts show that such a replacement is always possible.

By induction, we can replace every S_x with G_x for $x \geq i$, and since $S_x = G_x$ for $x < i$ anyway, let's replace those also. Now, we've replaced the entire optimal solution $S_{1..k}$ with $G_{1..k}$, and at this point if $k = l$, we'd have $S = G$ entirely, telling us that our greedy algorithm works. That has to be the case though, because if those k classrooms were enough for all n students, our greedy output would've stopped at G_k since the algorithm always stops at the first moment where all n students have their classrooms provided. This means that there'd never be an element G_{k+1}, which indicates that $k = l$.

Therefore, our greedy algorithm always works (outputs the minimal set of classrooms enough for n students). $\square$

4. (Cost-controlled COVID Classrooms BT) Let's call this algorithm MC, which stands for "minimum cost." It takes three arguments: n is the number of students, $C[1..m]$ is the list of capacities, and $E[1..m]$ is the list of expenses:

$$\text{MC}(n, C[1..m], E[1..m]) = \begin{cases} 0, \textbf{if } n \leq 0; \\ +\infty, \textbf{otherwise if } m = 0; \\ \min \begin{Bmatrix} \text{MC}(n - C[m], C[1..(m-1)], E[1..(m-1)]) + E[m], \\ \text{MC}(n, C[1..(m-1)], E[1..(m-1)]) \end{Bmatrix}, \\ \quad \textbf{otherwise} \end{cases}$$

This algorithm applies the "use it or lose it" strategy. More specifically, it takes advantage of the fact that, for any classroom c, c either contributes to the set of classrooms that's eventually provided to the students, or it doesn't. We let c be the *last* classroom in the list, which then reduces the problem size by one and creates the exact same problem. The two possible base cases are either there are no more students waiting to be assigned classrooms, which would cost no expense, or we run out of rooms to provide, which we assign a cost of infinity so it'd never be chosen as the final result (unless we don't have enough rooms to start with). Moreover, since the minimum is taken at each recursive level, the final result is definitely the minimum across all levels. Combining all these ideas, this algorithm is correct. □

5. (Cost-controlled COVID Classrooms DP, part a) The algorithm is as follows. Note that the argument n is the total number of students and C is an array with the m classrooms, each having two properties called `capacity`, the value of C_i, and `expense`, the value of E_i.

```
Algorithm: (DP) min_cost
  Input: int n, classroom C[1..m]
  Output: int

  int answers[1..n][0..m]
  for int i = 1 to n do
    answers[i][0] ← +∞
  for int i = 1 to n do
    for int j = 1 to m do
      int use_it ← C[j].expense
      if i > C[j].capacity then
        use_it ← use_it + answers[i - C[j].capacity][j - 1]
      int lose_it ← answers[i][j - 1]
      answers[i][j] ← min(use_it, lose_it)
  return answers[n][m]
```

5. (Cost-controlled COVID Classrooms DP, parts b/c)

(b) This algorithm follows the strategy of the backtracking algorithm from the previous problem, and we made it much more efficient by using the technique of memoization. In the DP algorithm, we use a two-dimensional array for memoizing, which is advantageous because it gives $O(1)$ run-time for both reading and writing data at any pair of indices. We want an element at indices (i, j) in the array to record the minimum cost of assigning i students into the classrooms in the sub-array $C[1..j]$. Since in the original backtracking algorithm, the recursive calls only ever involve smaller values of i and j (though they were called n and m in the previous problem), we calculate the results and fill in the array from smallest to greatest indices in both dimensions. By doing it in this order, we can guarantee that the value is valid when we do a lookup, since we only ever look up from smaller indices than the current i and j. We also added an extra column in the beginning of j's dimension to our advantage so we wouldn't get out of bounds when looking up column $(j - 1)$. This column also represents our base case: when there are more students to assign but rooms have run out. When the algorithm finishes, the value stored at index (n, m) in the memoizing array will have the final answer, the minimum cost of assigning n students into classrooms in $C[1..m]$. $\square$

(c) This algorithm has an $O(n \cdot m)$ run-time complexity and an $O(n \cdot m)$ space complexity (easy to see and prove from the code). $\square$