Monday, February 9, 2026 Instructor: Cynthia Bailey

CS106B Midterm Exam - Winter 2026
SOLUTIONS

Problem 1:

(a) (3pts) Write the output of the line marked Checkpoint 1 in this box:

Front `{ 5, 5, 6, 6, 1, 1 }` *Back*

(b) (3pts) Write the output of the line marked Checkpoint 2 in this box:

Bottom `{ 5, 5, 6, 6, 1, 1 }` *Top*

(c) (3pts) Write the output of the line marked Checkpoint 3 in this box:

Front `{ 4, 3, 2 }` *Back*

(d) (3pts) Write the output of the line marked Checkpoint 4 in this box:

`{ 1, 4, 6, 3, 5, 2 }`

Problem 2:

(a)

☐ `int closestForBudget(int budget, Grid<int>& prices) {`
☑ `int closestForBudget(int budget, Vector<int>& bestInRow) {`

```
// bestInRow is a better choice because it avoids an O(N) search across all the
// columns of the Grid
    for (int row = 0; row < bestInRow.size(); row++) {
        if (bestInRow[row] <= budget) {
            return row;
        }
    }
    return NOT_FOR_SALE;
}
```

SUID:

```
// If you chose the incorrect function header, you can still get a working
// implementation as follows:
int closestForBudget(int budget, Grid<int>& prices) {
    for (int r = 0; r < prices.numRows(); r++) {
        for (int c = 0; c < prices.numCols(); c++) {
            if (prices[r][c] <= budget) {
                return r;
            }
        }
    }
    return NOT_FOR_SALE;
}
```

(b)

```
void sellTicket(GridLocation seat, Grid<int>& prices, Vector<int>& bestInRow) {
    // NOTE THE ORDER OF THE CHECK MATTERS HERE
    if (!prices.inBounds(seat) || prices[seat] == NOT_FOR_SALE) {
        error("invalid seat");
    }
    int price = prices.get(seat);
    prices[seat] = NOT_FOR_SALE; // update before going to look for new best
    if (bestInRow[seat.row] == price) {
        int best = NOT_FOR_SALE;
        for (int col = 0; col < prices.numCols(); col++) {
            if (prices[seat.row][col] < best) {
                best = prices[seat.row][col];
            }
        }
        bestInRow[seat.row] = best;
    }
}
```

```
// Solution 1
Map<GridLocation, int> getGroupOptions(int size, Grid<int>& prices) {
    // Rubric Item 1 (initialize map)
    Map<GridLocation, int> result;
    for (int r = 0; r < prices.numRows(); r++) {
         // Rubric Item 2 (for loop over rows)
        for (int start = 0; start < prices.numCols(); start++) {
            // Rubric Item 3 (for loop over cols)
            int total = 0;
            for (int c = 0; c <= size; c++) {
                // Rubric Item 4 (for loop over the adjacent seats)
                if (c == size - 1) {
                    // Rubric Item 5 (updating the return map)
                    result[GridLocation(r, start)] = total;
                    break;
                }
                /*
                 * All locations in the streak must
                 * be in bounds and for sale.
                */
                if (!prices.inBounds(r, start + c)
                    || prices[r][start + c] == NOT_FOR_SALE) {
                    break;
                }
                // Rubric Item 5 (updating the total)
                total += prices[r][start + c];
            }
        }
    }
    // Rubric Item 6 (return map)
    return result;
}

// Solution 2
Map<GridLocation, int> getGroupOptions(int size, Grid<int>& prices) {
    Map<GridLocation, int> result;
    for (int r = 0; r < prices.numRows(); r++) {
        int numSeatsFound = 0;
        int totalPriceSoFar = 0;
        for (int c = 0; c < prices.numCols(); c++) {
            if (prices.get(r, c) != NOT_FOR_SALE) {
                // If this seat is valid, increment the counter and price.
                numSeatsFound++;
                totalPriceSoFar += prices.get(r, c);
                // Once the counter reaches size, we are ready to add
                if (numSeatsFound == size) {
                    // Note the column math here.
                    result[{r, c - size + 1}] = totalPriceSoFar;
                    // IMPORTANT: if you forget to decrement the counters,
```

SUID:

```
                    // you will miss overlapping groups!
                    numSeatsFound--;
                    totalPriceSoFar -= prices.get(r, c - size + 1);
                }
            } else {
                // IMPORTANT: if you don't reset the counters, then
                // the regions we find might not be adjacent.
                numSeatsFound = 0;
                totalPriceSoFar = 0;
            }
        }
    }
    return result;
}
```

Problem 3

```
// Solution 1

Set<string> signRescue(string plan, int nFlakes) {
    string soFar;
    Set<string> result;
    helper(plan, nFlakes, soFar, result);
    return result;
}

// Slight variation uses an additional parameter saying the desired length
// and the base case checks against that. (In that case, do not need to pass
// the nFlakes parameter.)
void helper(string plan, int nFlakes, string soFar, Set<string>& result) {
    if (plan.length() == nFlakes) {
        if (isAllowed(soFar)) {
            result += soFar;
        }
    }
    for (int i = 0; i < plan.length(); i++) {
        helper(plan.substr(0, i) + plan.substr(i + 1), nFlakes, soFar + plan[i],
result);
    }
}



// Solution 2
Set<string> signRescue(string plan, int nFlakes) {
     return signHelper(plan, nFlakes, "");
}
```

```
Set<string> signHelper(string plan, int nFlakes, string soFar) {
     if (plan.size() == nFlakes) {
          if (isAllowed(soFar)) {
               return {soFar};
          }
     }

     Set<string> result;

     for (int i = 0; i < plan.size(); i++) {
          string rest = plan.substr(0, i) + plan.substr(i + 1);
          result += signHelper(rest, nFlakes, soFar + plan[i], signs);
     }

     return result;
}
```

Problem 4:

(a) $O(N^2)$
(b) O(logN)
(c) O(N) (This tripped up a lot of folks—what I wanted you to notice was that the Vector is not being passed by reference, so it is expensive, O(N), to make a whole new copy of it. The constructor that takes an initial size is also O(N), but mostly I wanted you to remember why we always pass our Stanford library container objects by reference.)
(d) $O(N^2)$ (Each outer loop iteration incurs an O(N) inner for-loop cost, and there are N loop iterations)
(e) $O(N^2)$ (Same analysis—each recursive call incurs an O(N) for-loop cost, and there are N recursive calls)
(f) O(N) (just a whole bunch of O(N) operations one after the other, and they add up to O(N) because the constant coefficient goes away)