

CS106B Practice Midterm 1 Solutions

Problem 1 Stacks and Queues:

- (a) { 5, 5, 6, 6, 1, 1 }
- (b) { 5, 5, 6, 6, 1, 1 }
- (c) { 4, 3, 2 }
- (d) { 1, 4, 6, 3, 5, 2 }

Problem 2: BTS and BFS:

(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;
}

// 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;
    }
}
```

(c)

// 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,
                    // 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;
}

```

3. Big-O

- (a) $O(N^2)$
- (b) $O(\log N)$
- (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) $O(N)$ (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)