A2Q1

(b)

$M3 > M1 > M4 > M6 > M5 > M2 > M8 > M7$

1. Divide: $M3 > M1 > M4 > M6$ and $M5 > M2 > M8 > M7$
2. Divide: $M3 > M1$ and $M4 > M6$ and $M5 > M2$ and $M8 > M7$
3. Merge: Conflicts for $M3 > M1$, $M5 > M2$, and $M8 > M7$ (# = 3)
4. Merge: $M1 > M3 > M4 > M6$ and $M2 > M5 > M7 > M8$
   Conflicts for M3 > M2, M4 > M2, M6 > M2, M6 > M5 (# = 4)
5. Therefore # total conflicts is 7

A2Q2

```
Quads: enum {TR, BR, BL, TL}
(top-right, bottom-right, bottom-left, top-left)

       (i, j)
L.TR: {(0, 0), (1, 0), (1, 1)}
  or  X
      XX
L.BR: {(0, 0), (0, 1), (1, 0)}
  or  XX
      X
L.BL: {(0, 0), (0, 1), (1, 1)}
  or  XX
       X
L.TL: {(1, 0), (0, 1), (1, 1)}
  or   X
      XX

Inserts tile L in grid with position as top-left origin
The 'X' positions become new cavities
insertLInGrid(grid, L, position)
  for square in L:
    grid[position + square.i][position + square.j] = 'X'

grid: Grid of size n*n to tile with Ls
[0][0] being the initial cavity (in TL quad)
TileSquare(grid, n)
  if n = 1 return grid
  else
    insertLInGrid(grid, L.(grid.cavity_quad), (n/2)-1)
    TileSquare(grid.TR, n/2)
    TileSquare(grid.BR, n/2)
    TileSquare(grid.BL, n/2)
    TileSquare(grid.TL, n/2)

  return grid
```

(a)

Complexity

The algorithm has a simple recurrence relation of

$T(n) = 4T\left(\frac{n}{2}\right) + d$

Where d is constant time. Using Master Method, the time complexity is $\theta(n^2)$.

Correctness: Proof by Induction
Base case: We know $n \geq 2$. For n = 2, we have an initial cavity in `grid[0][0]`. We insert a top-left L (as defined above in pseudo-code) at position `[0][0]` which fills the other 3 squares in the grid. The square is tiled with Ls with the initial cavity.
Induction Hypothesis: Assume we can tile the grid an $(n-1)^2$ grid with L's.
Inductive Step: For n, we place a top-left L that creates cavities in each of the other 3 quadrants. Now each quadrant has a cavity and we can divide the grid into 4 sub-problems of side $(n-1)$. By inductive hypothesis, we saw that the sub-problems are solvable. Therefore the grid is tiled with L's.

(b)

Above code would not need any modification except for the fact that the cavity can exist at any location, not just `[0][0]`. `grid.cavity_quad` should always give us the quadrant in which the cavity lives in.

A2Q3

```
NOT_EQ: Not equal identifier, does not exist in array C
count(A, elem): returns the number of times elem occurs in
array A

C: Array of results of n Computers
Returns the popular result if it is popular, NOT_EQ otherwise
PopularResult(C, n):
  if (n = 1) {
    return C[1]
  } else {
    left  = PopularResult(C[1..(n/2)], n/2)
    right = PopularResult(C[(n/2 + 1)..n], n/2)
    return left if left = right

    leftFreq  = count(C, left)
    rightFreq = count(C, right)

    if leftFreq > n/2 return left
    else if rightFreq > n/2 right right
    else return NOT_EQ
  }
```

Complexity

The algorithm has a simple recurrence relation of

$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

since we divide the array into two halves at each step and $O(n)$ is the complexity of the calls to `count()` method.

Using Master Method, the time complexity is $\theta(n \log n)$.

Correctness

We convert the array into a kind of a binary tree and move up the popular result at each level of the tree. We determine if an element is a popular result if:

1. The element is equal to the immediate sibling in the tree
2. The element occurs more than n/2 times within the nodes of its subtree

Say a popular result occurs in the first half and one element of the second half of C. We will obviously get the popular result from the left subtree of C via point (1). But we will get NOT_EQ through the right subtree. However when we compare whether the results of the left and right subtrees and find them to not be equal, we check the frequency of the result of left subtree in the whole C and find that it exists more than n/2 times. Therefore it is a popular result.

A2Q4

(a)

$T_1 = (10, 6), T_2 = (7, 8), T_3 = (4, 2)$

Greedy Strategy 1: Increasing c_i
$S = [T_3, T_2, T_1]$

Test	**Completion Time**
T_3	4 + 2 = 6
T_2	4 + 7 + 8 = 19
T_1	4 + 7 + 10 + 6 = 27
max()	27

Greedy Strategy 2: Decreasing c_i
$S = [T_1, T_2, T_3]$

Test	**Completion Time**
T_1	10 + 6 = 16
T_2	10 + 7 + 8 = 25
T_3	10 + 7 + 4 + 2 = 23
max()	25

Greedy Strategy 3: Increasing m_i
$S = [T_3, T_1, T_2]$

Test	**Completion Time**
T_3	4 + 2 = 6
T_1	4 + 10 + 6 = 20
T_2	4 + 10 + 7 + 8 = 29
max()	29

Greedy Strategy 4: Decreasing m_i
$S = [T_2, T_1, T_3]$

Test	**Completion Time**
T_2	7 + 8 = 15
T_1	7 + 10 + 6 = 23
T_3	7 + 10 + 4 + 2 = 23
max()	**<u>23</u>**

Greedy Strategy 5: Increasing $c_i + m_i$
$S = [T_3, T_2, T_1]$

Test	**Completion Time**
T_3	4 + 2 = 6
T_2	4 + 7 + 8 = 19
T_1	4 + 7 + 10 + 6 = 27
max()	27

Greedy Strategy 6: Decreasing $c_i + m_i$
$S = [T_1, T_2, T_3]$

Test	**Completion Time**
T_1	10 + 6 = 16
T_2	10 + 7 + 8 = 25
T_3	10 + 7 + 4 + 2 = 23
max()	25

Most optimal greedy strategy out of the 6: #4 Decreasing m_i

(b)

Suppose G is the greedy solution and S is an optimal solution. Let T_i and T_j exist in S where $1 \leq i < j \leq n$ and $j = i + 1$. Let t be the time taken by an optimal solution.

We know,
$CompletionTime(i)$ = $CompletionTime(i-1) + c_i + m_i$
$CompletionTime(j)$ = $CompletionTime(i-1) + c_i + c_j + m_j$

If we swap T_i and T_j in S,
$CompletionTime(i)$ = $CompletionTime(i-1) + c_j + c_i + m_i$
$CompletionTime(j)$ = $CompletionTime(i-1) + c_j + m_j$

If $m_i \leq m_j$, we arrive at another optimal solution. By a sequence of swaps of this type, the optimal solution can be transformed into the greedy schedule, preserving optimality at each step.

A2Q5

```
D: Sorted array of distances of n hotels
L: Max possible travel distance / day
HotelsToStopAt(D, L, n) {
  i = 1
  traveled = 0
  while (i != n) {
    nextDistance = D[i+1] - D[i]
    if ((nextDistance + traveled) >= L) {
      report i
      traveled = 0
    } else {
      traveled += nextDistance
    }
    i++
  }
}
```

Complexity
We loop through the array once in the while loop. All other operations are constant time. Therefore the complexity is $\theta(n)$.

Correctness
The greedy algorithm selects the farthest hotel possible within the possible limit L. A more optimal solution is not possible as that would violate the limit L. All optimal solutions would either be the greedy solution or select an earlier hotel. Therefore at each iteration, we are either at par or ahead of the optimal solution.