Homework 1

Problem 5

1. a. A is O, o of B.
   b. A is none of B.
   c. A is Ω, ω of B.
   d. A is O, Ω, Θ of B.
   e. A is O, Ω, Θ of B.

2. a. The input is an integer sequence M of length n, and the output is also an integer sequence but of length 2 with the first element being the time to approach and the second element being the time to disconnect.

```
int[] range(int[] M) {
    int n = M.length;
    if (n == 1) return new int[] { 0, 0 };
    // divide
    int[] lt = range(M[0...n/2]); // left division
    int[] rt = range(M[n/2...n]); // right division
    // search for the spanning subsequence from left to right division
    int l = n/2, r = n/2;
    while (l > 0 && M[l-1] < M[l]) l--;
    while (r < n-1 && M[r+1] > M[r]) r++;
    // return the range with the highest changing score
    int[][] candidates = new int[][] { lt, new int[] { l, r }, rt };
    int i = argmax(M[lt[1]]-M[lt[0]], M[r]-M[l], M[rt[1]]-M[rt[0]]);
    return cnadidates[i];
}
```

   b. The recurrence formula is written as $T(n) = 2T(\frac{n}{2}) + n$. The "conquer" step may search the entire sequence (in the worst case) for a spanning subsequence, and therefore, the time complexity is n, or $\Theta(n)$. According to the master theorem, $T(n) = \Theta(n \log n)$.

Problem 6

1. (a)(b)(c) are solved using the master theorem.
   a. $f(n) = n = o(n^{\log_2 4}) \implies T(n) = O(n^2)$.
   b. $f(n) = \frac{n}{\log n} = \Theta(n^{\log_2 2}) \implies T(n) = O(f(n) \log n) = O(n)$.

c. $f(n) = n^3 = \omega(n^{\log_3 9}) \implies T(n) = O(n^3)$.

d. Time complexity at the k-th level of recursion tree is $(\frac{1}{5} + \frac{7}{10})^k n = (\frac{9}{10})^k n$. Total time complexity is $T(n) \leq (1 + \frac{9}{10} + (\frac{9}{10})^2 + \ldots)n = 10n = O(n)$.

e. At the k-th level of recursion tree, there are $n^{\frac{1}{2}+\frac{1}{4}+\ldots+\frac{1}{2^k}}$ nodes, each of size $n^{\frac{1}{2^k}}$. The number of levels, K, is obtained from $n^{\frac{1}{2^K}} = c \implies K = (\log\log n - \log\log c)/\log 2 = O(\log\log n)$. Therefore, total time complexity is $T(n) \leq \sum_{k=0}^{O(\log\log n)} n^{\frac{1}{2}+\frac{1}{4}+\ldots+\frac{1}{2^k}} n^{\frac{1}{2^k}} = \sum_{k=0}^{O(\log\log n)} n = O(n \log\log n)$.

2. The time complexity of the problem is $O(nm)$ if there exists an $O(nm)$-time algorithm that solves the problem; otherwise, we disapprove it. The truth is that there indeed exists such an $O(nm)$-time algorithm as explained below.
Given a binary matrix $M \in \mathbb{R}^{n\times m}$. The algorithm relies on an auxiliary matrix $A \in \mathbb{R}^{n\times m}$ with each element representing the size of the square submatrix which contains only 1s, assuming that element is the rightmost and bottommost element of that submatrix. The auxiliary matrix A is constructed in a way similar to dynamic programming. The size of the largest square submatrix is the maximum value of A, and the rightmost and bottommost element of that submatrix is the position of the maximum value of A.

```
int[][] solver(int[][] M) {
    int n = M.length;
    int m = M[0].length;
    int[][] A = new int[n][m]; // create the auxiliary matrix
    for (int i = 0; i < n; i++)
        A[i][0] = M[i][0];
    for (int j = 0; j < m; j++)
        A[0][j] = M[0][j];
    for (int i = 1; i < n; i++)
        for (int j = 1; j < m; j++)
            if (M[i][j] == 0)
                A[i][j] = 0;
            else
                A[i][j] = min(A[i][j-1], A[i-1][j], A[i-1][j-1]) + 1;
    // calculate the properties of the largest square submatrix
    int size = max(A); // max returns the maximum value
    int[] position = argmax(A); // argmax returns its row and column
    return new int[][] { new int[] { size }, position };
}
```

The time complexity of each step are listed as follows: first for loop: $O(n)$; second for loop: $O(m)$; third (nested) for loop: $O((n-1)(m-1))$. The if-else statement in the nested for loop only takes constant time independent of the matrix size. Finally, both max and argmax function take $O(nm)$ to examine every entry in the auxiliary matrix to obtain the value and position of the maximum. Therefore, the total time complexity is $O(n) + O(m) + O((n-1)(m-1)) + O(nm) + O(nm) = O(nm)$.