Problem 1.

a. We consider the algorithm presented the lecture slides.

(i)
1. `power(b, 79)`
2. `power(b, 78)`
3. `power(b, 39)`
4. `power(b, 38)`
5. `power(b, 19)`
6. `power(b, 18)`
7. `power(b, 9)`
8. `power(b, 8)`
9. `power(b, 4)`
10. `power(b, 2)`
11. `power(b, 1)`
12. `power(b, 0)`

(ii) The worst case occurs when each division of n by 2 results in an odd number. When this occurs, each recursive call that divides n by 2 results in another recursive call that subtracts 1 from n. So, at most, we need $\lceil 2\log(n) \rceil$ recursive calls. Note that the $\log(n)$ term appears because when n is even, we divide by 2.

We can derive a worst case by starting from 0, adding 1, then repeating: multiply by 2, add 1 until we get some value greater than 50. We have $0 \to 1 \to 2 \to 3 \to 6 \to 7 \to 14 \to 15 \to 30 \to 31 \to 62 \to 63$ (12 total calls). Hence a worst-case scenario occurs when $n = 63$ ($\lceil 2\log(62) \rceil = 12$).

b. Let m be the smallest power of 2 that is greater than n. We can pad the input matrices with zeroes until they become $m \times m$ matrices at which point we can perform matrix multiplication using Strassen's algorithm. This is done by adding $m - n$ zeroes to the end of each row and $m - n$ zeroes to the end of each column (including the new columns formed from adding to each row). We then extract the original matrix after application of Strassen's algorithm by removing the added rows and columns.

For example, let's say our input matrices are 30×30. In this case, 32 is the smallest power of 2 that is greater than 30. We would pad the input matrices with zeroes such that they become 32×32 matrices, at which point we apply Strassen's algorithm. This is done by adding two zeroes to the end of each row and two zeroes to the end of each column. After applying Strassen's algorithm, we remove the added rows/columns from the resultant matrix.

Problem 2.

a. We consider the algorithm presented in problem 5 of problem set #4.

(i) The schedule for each day is shown below:

Table 1: Round-Robin tournament schedule for 8 teams.

Team	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7
t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
t_2	t_1	t_4	t_3	t_6	t_7	t_8	t_5
t_3	t_4	t_1	t_2	t_7	t_8	t_5	t_6
t_4	t_3	t_2	t_1	t_8	t_5	t_6	t_7
t_5	t_6	t_7	t_8	t_1	t_4	t_3	t_2
t_6	t_5	t_8	t_7	t_2	t_1	t_4	t_3
t_7	t_8	t_5	t_6	t_3	t_2	t_1	t_4
t_8	t_7	t_6	t_5	t_4	t_3	t_2	t_1

(ii) The running time of the algorithm is given by the recurrence $T(n) = 2T(n/2) + cn^2$ where c is some positive constant.

We apply the master theorem. We have $a = 2$, $b = 2$, and $f(n) = cn^2$. We have that $\log_b(a) = 1$, so $n^{\log_b(a)} = n$. We see that $f(n) \in \Omega(n^{1+\epsilon})$ for $\epsilon = 1.1$, so case 3 applies. We can also confirm $af(n/b) \leq \delta f(n)$ for some constant $\delta < 1$ for all sufficiently large n. We have that

$$\begin{aligned} af(n/b) &= 2c\left(\frac{n}{2}\right)^2 \\ &= \frac{cn^2}{2}. \end{aligned}$$

Let us select $\delta = 3/4$. Then clearly $af(n/b) \leq \delta f(n)$ for all sufficiently large n. As a result, we conclude $T(n) \in \Theta(n^2)$.

(iii) The algorithm cannot be generalized for any even n such that it requires at most $n - 1$ days. This is because if n is not a power of 2, then at some point, a recursive call will have an odd number of teams to handle. We provide a counterexample.

Let's consider the case where we have $n = 6$ teams. Our algorithm can have the first half of the teams (t_1, t_2, t_3) play the second half of the teams (t_4, t_5, t_6) within 3 days:

Table 2: Round-Robin tournament schedule for 6 teams (days 3 to 5).

Team	Day 3	Day 4	Day 5
t_1	t_4	t_5	t_6

t_2	t_5	t_6	t_4
t_3	t_6	t_4	t_5
t_4	t_1	t_3	t_2
t_5	t_2	t_1	t_3
t_6	t_3	t_2	t_1

Now that days 3 to 5 are scheduled, in order to satisfy that we don't require more than $n-1=5$ days, we need teams t_1, t_2, t_3 to play a tournament between themselves and teams t_4, t_5, t_6 to play a tournament between themselves within two days. However, this is impossible for either set of teams: on any given day, at least one team will not be able to play since there are an odd number of teams. This means we need at least 3 days to schedule a tournament between the teams (for example, if t_1 doesn't play on the first day, it needs at least 2 more days to play t_2 and t_3). This shows we require more than $n-1$ days.

b. We consider the algorithm presented in problem 8 of problem set #4.

(i) **Cup 3:** We choose $l_3 = n^{2/3}$. We test from step l_3, then $2l_3$, then $3l_3$ and so on up until n/l_3. If the cup breaks on step $i+1$, we know the last step at which the cup doesn't break is between il_3 and $(i+1)l_3 - 1$. There are at most n/l_3 experiments.

Cup 2: We then choose $l_2 = \sqrt{l_3}$. We test from step $il_3 + l_2$, then $il_3 + 2l_2$, then $il_3 + 3l_2$ and so on up until l_3/l_2. If the cup breaks on step $j+1$, we know the last step at which the cup doesn't break is between $il_3 + jl_2$ and $il_3 + (j+1)l_2 - 1$. There are at most l_3/l_2 experiments.

Cup 1: Lastly, we test from step $il_3 + jl_2 + 1$, $il_3 + jl_2 + 2$, $il_3 + jl_2 + 3$, and so on up until l_2. Let's say the cup breaks on step $r+1$. This means the last step at which the cup doesn't break is $il_3 + jl_2 + r$. There are at most l_2 experiments.

Analysis: In the first step, there are at most n/l_3 experiments. In the second step, there are at most l_2/l_3 experiments. In the last step, there are at most l_2 experiments. Hence, the total number of experiments is given by

$$\begin{aligned} T_3(n) &= \frac{n}{l_3} + \frac{l_3}{l_2} + l_2 \\ &= \frac{n}{n^{2/3}} + \frac{n^{2/3}}{\sqrt{l_3}} + \sqrt{l_2} \\ &= n^{1/3} + \frac{n^{2/3}}{\sqrt{n^{2/3}}} + \sqrt{n^{2/3}} \\ &= n^{1/3} + n^{1/3} + n^{1/3} \\ &\in O(\sqrt[3]{n}) \\ &\in o(\sqrt{n}). \end{aligned}$$

(ii) **Cup 4:** We choose $l_4 = n^{3/4}$. We test from step l_4, then $2l_4$, then $3l_4$ and so on up until n/l_4. If the cup breaks on step $p+1$, we know the last step at which the cup doesn't break is between pl_4 and $(p+1)l_4 - 1$. There are at most n/l_4 experiments. We can then apply the solution described in the previous question. We would simply start with $l_3 = l_4^{2/3}$. We would also start the tests for cup 3 at pl_4.

Analysis: The total number of experiments is given by

$$\begin{aligned}
T_4(n) &= \frac{n}{l_4} + \frac{l_4}{l_3} + \frac{l_3}{l_2} + l_2 \\
&= \frac{n}{n^{3/4}} + \frac{n^{3/4}}{(n^{3/4})^{2/3}} + \frac{(n^{3/4})^{2/3}}{\sqrt{(n^{3/4})^{2/3}}} + \sqrt{(n^{3/4})^{2/3}} \\
&= n^{1/4} + \frac{n^{3/4}}{n^{2/4}} + \frac{n^{2/4}}{n^{1/4}} + n^{1/4} \\
&= n^{1/4} + n^{1/4} + n^{1/4} + n^{1/4} \\
&\in O(\sqrt[4]{n}) \\
&\in o(\sqrt[3]{n}).
\end{aligned}$$

Problem 3.

We consider the algorithm presented in problem 6 of problem set #4.

a. The algorithm outputs 5.

b. The algorithm outputs 5.

c. The algorithm travels down the array A and sums the numbers it encounters until it finds the first negative number at index i. Let us call this sum S_1. Once it finds the first negative number, the algorithm finds the sum of the two negative numbers by adding $A[i]$ and $A[i+1]$ (using the fact that the two negative numbers are consecutive). Let us call this sum N. After this, the algorithm starts at index $i+2$ and sums the remaining elements in the array. Let us call this sum S_2. The algorithm then returns the maximum of S_1, $S_1 + S_2 + N$, and S_2.

   Since the algorithm iterates over each element of the array once, its running time is in $O(n)$. This is faster than the algorithm presented in the problem set: the running time of the algorithm in the problem set is in $O(n \log(n))$.

Problem 4.

We define our input array as A and the value we wish to search for as v. Our algorithm returns `true` if v is found in A and `false` otherwise. Our algorithm begins in the top right corner (i.e. index $i = 1$, $j = m$). We repeat:

1. Compare the element in this cell (i.e. $A[i][j]$) with v. If it is equal, we return `true`.
2. If the element in this cell is less than v we increment i. If $i > n$ then we return `false`. Otherwise, we repeat starting from step 1.
3. Otherwise we decrement j. If $j < 1$ then we return `false`. Otherwise, we repeat starting from step 1.

The pseudocode is provided below. It is written in Lua. Note that arrays in Lua begin at 1. Furthermore, the `local` keyword is used to define a new variable.

```
function IsInMatrix(A, n, m, v)
    local i = 1
    local j = m

    while i <= n and j >= 1 do
        if A[i][j] == v then
            return true
        elseif A[i][j] < v then
            i = i + 1
        else
            j = j - 1
        end
    end

    return false
end
```

The worst case occurs when the target value is in the bottom left corner. We would need to check n cells in the vertical direction and m cells in the horizontal direction. As a result, we can conclude the running time of the algorithm is in $O(n + m)$.

The best case occurs when the target value is in the top right corner in which case the algorithm returns immediately. This would lead to the running time being in $O(1)$.