Exercise 1

Part 1

Each palindrome can have either an odd number or an even number of characters. First, we consider the case of odd length, i.e., $P = (w_i, ..., w_h, ..., w_j)$ where $w_{h-1} = w_{h+1}$, $w_{h-2} = w_{h+2}$, and so on. We can take one character at a time and choose it as the center of a palindrome. Starting from the center w_h, we check if the two adjacent characters w_{h-1} and w_{h+1} are equal, and we continue in both directions until this condition of equality is satisfied and we don't reach the ends of the string. If the length of this new palindrome is greater than the length of the previous ones, we store it as the best palindrome found so far. Now, we consider the case of even length, i.e., $P = (w_i, ..., w_{h-1}, w_h, ..., w_j)$ where $w_{h-1} = w_h$, $w_{h-2} = w_{h+1}$, and so on. We execute the same procedure with only one difference: the center of the palindrome must be a couple of equal characters w_h and w_{h-1}. The time complexity of this algorithm is $O(n^2)$ because it has two simple nested loops. The string has n characters and, for each one of these, we analyze all the other characters in the worst case.

Part 2

We can solve this problem using the dynamic programming technique. In particular, the algorithm uses the following recursive function:

$$OPT(i,j) = \begin{cases} 0 & \text{if } i = 0 \lor j = 0 \\ 1 + OPT(i-1, j-1) & \text{if } w_i = w_j \\ max\{OPT(i-1,j), OPT(i,j-1)\} & \text{if } w_i \neq w_j \end{cases}$$

The two parameters i and j are indices that iterates respectively along the string w and the reverse string $w^R = (w_n, ..., w_1)$. In the recursive function, we compare two characters at a time and we have two non-trivial cases. If $w_i = w_j$, we know that the two characters belong to a palindrome: we add 1 to the value, and we apply the recursive step on the remaining characters. If $w_i \neq w_j$, then we search the longest palindrome between the subproblems found by removing w_i and w_j. The time complexity of the algorithm can be greatly improved with memoization. To avoid computing the same results over and over again, we can store them using a $(n+1)\times(n+1)$ matrix M, in which the first row $M[0][j]$ and the first column $M[i][0]$ represent the base cases. In this way, it is only needed to fill up the matrix in a bottom-up fashion, and this takes $O(n^2)$. The longest palindrome can be constructed by exploiting the matrix: starting from $M[n][n]$, we decrement index j until $M[i][j] = M[i][j-1]$ and then decrement index i until $M[i][j] = M[i-1][j]$; at this point, we have $M[i][j] = 1 + M[i-i][j-i]$, we decrement both indices and take w_i as one character of the palindrome.

Exercise 2

We have to construct a directed graph $G = (V, E)$. Each investor $i \in I$ and each founder $f \in F$ can be represented as a vertex of our graph. Adding two vertices s and t, we have $V = I \cup F \cup \{s\} \cup \{t\}$.

The relation *good pairs* $P \subseteq I \times F$ can be modeled as a set of edges from investors to founders. In particular, there exists an edge $e = (i, f) \in E$ if and only if $(i, f) \in P$. These edges will have capacity $c(e) = 1$. Then, for each investor we add an edge (s, i) and for each founder we add an edge (f, t). All these edges will have capacity $c(e) = 2$.

An *st*-flow f is a function that satisfies the capacity and the flow conservation constraints. The edge $e = (i, f)$ will carry one unit of flow if i and f are neighbors in the seating arrangement. It's possible to prove that the maximum flow M saturates all the edges (s, i) and (f, t) if and only if there exists a seating arrangement with only good pairings

$$M = \sum_{e \text{ out of } s} c(e) = \sum_{e \text{ into } t} c(e)$$

Let's prove the first part of the claim. If the maximum flow saturates one edge (s, i), this means that there are two edges (i, f) out of i carrying one unit of flow (flow conservation constraint). In other words, the neighbors of i are both good founders. In the same way, if the maximum flow saturates one edge (f, t), this means that there are two edges (i, f) into f carrying one unit of flow, and the neighbors of f are both good investors. It is possible to extend these considerations to all the edges (s, i) and (f, t).

Now we can prove the second part of the claim. If the neighbors of i are both good founders, this means that two edges (i, f) out of i will carry one unit of flow. This implies that the edge (s, i) is saturated (flow conservation constraint). In the same way, if the neighbors of f are both good investors, this means that two edges (i, f) into f will carry one unit of flow, and the edge (f, t) is saturated.

It's simple to see that the equality doesn't hold if one of the following condition is true:

- $|I| \neq |F|$
- $|\{i : (i, f) \in P\}| < 2$
- $|\{f : (i, f) \in P\}| < 2$

Exercise 3

Given a list of n projects $p \in P$, we can work on each if and only if, once ordered, we have $\forall\, i < n$, $C + \sum_{k=0}^{i} b_i \geq c_{i+1}$, where $b_0 = 0$. The algorithm must find the order in which it is best to take the n projects. To do this task, the algorithm needs three main steps:

- place all the projects p with $b_i \geq 0$ at the beginning of the order
- sort the projects with $b_i \geq 0$ in ascending order with respect to c_i
- sort the projects with $b_i < 0$ in descending order with respect to $b_i + c_i$

In other words, we can split the list into two sublists using the value of b_i and then sort the sublists independently. After these steps, it is not guaranteed the order meets the requirement: we have to scan the projects in the given order to see if we can accomplish them all. It takes $O(n)$ and step 1 does the same. The time complexity is bounded by step 2 and step 3, that take $O(n \log n)$ with an efficient sorting algorithm. It's possible to prove the correctness of this algorithm taking into account the three steps one at a time.

Step 1

We can consider two projects such that $b_i \geq 0$ and $b_j < 0$. If the actual score C (eventually modified by other projects before i and j) is less than c_i, then we can't find a right order. The project i doesn't satisfy the requirement, and it will not do so even if taken after j because $C + b_j < C < c_i$. Now, let's consider the case in which $C \geq c_i$. If $C < c_j$, we must take i before j. Now it's crucial to see that, even if $C \geq c_j$, we have to choose i before j: $C + b_j$ might be less than c_i, while $C + b_i > C \geq c_j$. We have proved that we always want to take i before j when $b_i \geq 0$ and $b_j < 0$.

Step 2

In this step we consider two projects such that $b_i \geq 0$ and $b_j \geq 0$ and $c_i < c_j$. We have three possible cases. If $C < c_i$, there is no solution to our problem, because i and j doesn't satisfy the requirement. If $c_i \leq C < c_j$, we are forced to work on i first. If $C \geq c_j$, we can choose which project to take before the other. Therefore we can sort the projects in ascending order with respect to c.

Step 3

Finally, we consider two projects such that $b_i < 0$ and $b_j < 0$. We can find a right order only if $C \geq max c_i, c_j$, otherwise we cannot work on at least one project. Let $c_i + b_i > c_j + b_j$, that is, $c_i - b_j > c_j - b_i$.

If $C < c_j - b_i$, we cannot work neither on i or j by definition: $C + b_i < c_j$ and $C + b_j < c_i$. If $c_j - b_i \leq C < c_i - b_j$, we are forced to work on i first. If $C \geq c_i - b_j$, we can choose which project to take before the other. Therefore we can sort the projects in descending order with respect to $c + b$.

Exercise 4

Part 1

The cure c_i is the most effective one if and only the related dose a_i is minimal, and we know that $\forall\, i,\ a_i \leq d$. Let $T(b) = |\{c_k : a_k \leq b\}|$, where $b \in \mathbb{N}^+$. By definition $T(d) = n$. In the beginning, we know the right answer is between the value d and 1. We call them respectively upper bound ub and lower bound lb. The algorithm considers the ceil of the mean cm between the two extreme values and tests the cures with this new value. If $T(cm) > 0$, we know that there is at least one cure such that cm units will kill the virus. The value cm replaces the upper bound and we repeat the same step with the interval $[lb, cm]$. If $T(cm) = 0$, we know that cm units are not sufficient. The value cm replaces the lower bound and we repeat the same step with the interval $[cm, ub]$. The algorithm stops when $lb + 1 = ub$: in this case, we know exactly the optimal value a_i below which no cure will be effective. There can be more than one cure that is optimal. The number of steps is $O(\log d)$ because we are progressively halving the width of the interval ($d \to d/2 \to ... \to 1$). At each step, we test all the n cures in the worst case. Therefore the algorithm takes $O(n \log d)$.

Part 2

The randomized algorithm starts picking one cure c_i and executes the same steps as the deterministic algorithm. In this way, we can find the minimal a_i for this first cure. Then it picks at random a second cure c_j and tests it with the value $a_i - 1$. If the test fails, then $a_j > a_i$ and we can immediately reject the cure c_j. If the test succeeds, we can reject c_i and execute the steps of the deterministic algorithm on the interval $[1, a_i - 1]$ to find the optimal a_j. The algorithm repeats this procedure until all cures have been tested at least one time.

If you pick a random c_j, the probability that a_j is minimum is $1/(n-k)$, where k is the number of cures already rejected. After the first test needed to decide which cure has to be rejected , the number of test for c_j is

$$T_j = \begin{cases} 0 & p = 1 - \frac{1}{n-k} \\ O(\log d) & p = \frac{1}{n-k} \end{cases}$$

The total number of tests is $T = T_0 + T_1 + ... + T_{n-1}$. Using the linearity of expectation, we can compute the expected value

$$E[T] = \sum_{k=0}^{n-1} \frac{O(\log d)}{n-k} = O(\log d) \sum_{i=1}^{n} \frac{1}{i} = O(\log d) H(n) = O(\log n \log d)$$

At this number, we must add the n tests needed to execute the rejection condition for all the cures.