Question 1

Performance on this question was mixed. Relatively few students achieved full marks across the board, but most who attempted it received the majority of the marks for part 1.1, with students struggling more with 1.2.
The most common mistake was students submitting an $O(n^3$ solution for part 1.2 and incorrectly analysing it as $O(n^2)$. These students received 0 for part 1.2 and a small penalty (typically 2 marks) for 1.1 for incorrect complexity analysis.
2 main approaches, and a few other less common approaches were used. Instead of discussing each sub question, I will discuss both sub questions together, broken down by approach.

- **Approach 1:** $opt(i)$
  - this is the standard approach given in solutions (see solutions). This was the most common approach, most students who used it relatively well accross both sub-questions.
  - other than mis-assessing time complexity as $O(n^2)$ without pre-processing, the most common error in 1.1 was failing to account for $opt(0)$ case. These students made their base case $opt(1) = 1 + A[1$ and had their recursion not have a way of accounting for their being no previous pillar, i.e. the entire bridge is one span.
  - another common error in this approach was a cyclic dependency, where students had $opt(i)$ in some way depend on already knowing $opt(i)$.
  - in part 1.2 many students stated that we should precompute maxima and/or spans, but did not demonstrate how this could be done in $O(n^2)$ time. This is neccesary, as naively filling in that table using the max equation would take $O(n^3)$ time and not provide any overall saving, even if the recursion is done more efficiently.
- **Approach 2:** $opt(i, j)$
  - the next most common approach was to set the subproblems as a span from i to j and solve subproblems in order of increasing $j - i$ distance. not everyone got the recursion here correct. The recursion that DOES work in this case is

    $$opt(i, j) = minmin(opt(i, k) + opt(k, j)), (j - i)^2 + max(A[i...j])$$

    with the first term referring to the cost of splitting the current span in two, and the latter term referring to keeping it all in one span. Each side can be computed in $O(n)$ and there are $O(n^2)$ calls to opt.
  - the most common issue with this approach was that there is no way to transform this approach into an $O(n^2)$ solution, so most students who used this approach scored 0 or very badly on 1.2.
  - some students also provided 2 seperate recursive equations, without minimising between them, which is not correct
  - finally there were some students who had errors in their recursion, giving an incorrect result.
- **Approach 3: Dijkstra's Algorithm**
  - this approach was significantly less common, but is interesting and worth noting. Some students did the preprocessing similar to the official solution, creating a graph of span costs for every possible span, and then transformed this into an adjacency matrix, with edges as span costs, and nodes as locations on the bridge. They then used Dijkstras algorithm to find the minimum cost path between 0 and n.
  - this approach is isomorphic to approach one with the precomputation, but takes a different approach to finding the optimal final cost

- **Greedy**
  - the main INCORRECT approach used was a greedy solution. This usually involved starting with a version of the bridge with no pillars, then greedily adding 1 pillar at a time that minimised cost after its addition, or doing the opposite, by removing one pillar at a time. The problem with these approaches, although appealing and seeming to work on many sample inputs, is that globally optimal solutions are not always reachable by locally optimal solutions.
    For example, consider the counterexample $[9, 1, 1, 9]$. I will not work through the maths here, but essentially, there is no solution with two spans with lower total cost than the solution with zero spans, but the optimal solution involves splitting the bridge into 3 spans. Any solution that involves only adding a pillar if it lowers total cost will never find the 3 span solution.
  - although these approaches seem appealing, and some students gave very well laid out and reasoned approaches, no one who used a greedy approach achieved more than 9/20 as the approach will not be correct in general.

Question 2

This question is very straightforward and is similar to the 'moving chess piece' problem discussed in the tutorial. Students who attempted the tutorials managed to solve it easily. A considerable number of students got full marks for this question. A large proportion of student received 26 - 28 out of 30 and the main reason for these students to lose 2 - 4 marks was the incompleteness of their base cases. We expected the main parts of a dynamic programming solution; namely the sub-problem definition, base cases, recurrence relation, the final answer, to be stated explicitly. Also, we expected the students to justify the correctness of their answers by explaining the segments in the recurrence and showing how the recurrence leads to the intended output. Correctness of a dynamic programming solution by nature is very reliant on **the correctness and the preciseness of the recurrence relation**. So, students who provided a different recurrence (even slightly different) would notice that a significant proportion of their marks been taken off. Nevertheless, this part was marked leniently. Therefore, be aware that a mark review request could adjust your mark downwards. However, if you strongly believe that your mark is incorrect, you are encouraged to submit a review request with a clear explanation of the reason for your review request.

2.1

- Very few students attempted this part separately.

2.2

- Almost all the students wanted this part to be evaluated against 2.1 as well.

Question 3

We begin with an exemplar response that is imperfect but reflects the general expectations of this question. Note that calculating the sum given on page 1 reflects the $O(n^2 * k)$ solution.

Solution

Begin by assuming that all the hungers are ordered without loss of generality. That meaning h_1 is the lowest hunger in the queue and h_N is the greatest hunger.
As we will see, we won't need to actually reference any of the exact h_i values, so actually sorting them serves no purpose. Thus, it is sufficient to simply assume (or imagine renumbering such that) all hungers are already sorted.
Also note that because all hungers are distinct, $h_{i-1} < h_i < h_{i+1}$ for all $i \in [1..n]$.
Subproblems: For each $1 \leq i \leq n$ and $0 \leq A \leq k$, we solve the subproblem $P(i, A)$:
What is $\text{opt}(i, A)$, the number of arrangements of a queue comprising the i least hungry people that give a total annoyance in the queue of A?

Base case: There are a few base cases to consider:

- When $i \leq 0$ there are no people in the queue and thus no possible arrangements. That is, $\text{opt}(i, A) = 0$ for $i \leq 0$.
- Similarly, there is no way to have negative annoyance in the queue, so $\text{opt}(i, A) = 0$ for $A < 0$.
- When there is exactly 1 person in the queue, there is only one possible arrangement and it results in no annoyance as there is no one to annoy or be annoyed by the 1 person. So, $\text{opt}(1, 0) = 1$ and $\text{opt}(1, A) = 0$ for $A \geq 1$.

Recurrance: Since we are adding people to the queue in increasing order of hunger, we know that, for the i^{th} person being added, $h_j < h_i < h_k$ for all $j < i < k$. That means that the i^{th} person will be annoyed by all the people already in the queue (with hunger $h_j < h_i$) they are placed behind of and will never be annoyed by any people (with hunger $h_k > h_i$) added after them.
A corollary of this, is that the $i - 1$ people already in the queue before i is added will never be annoyed by the addition of i.
In all, this means that adding the i^{th} person will increase the total annoyance by the number of people they are placed behind and this is the most annoyed this person will get even as more people are added. Thus, it is sufficient to only consider the annoyance of the i^{th} person when they are added to the queue.

In a queue of $i - 1$ people, the i^{th} person could be added in one of i positions. That is, they could be placed behind anywhere from 0 people, if placed at the front of the queue, to $i - 1$ people, if placed at the back of the queue.

In all, this means that adding the i^{th} person introduces anywhere from 0 to $i - 1$ annoyance to the existing queue and that this is the annoyance of person i even after all other people have been added.

We can now develop a recurrance based on this fact. Person i can be added to a queue of length $i - 1$ and annoyance $A - j$ for $j \in [0..i - 1]$ and create one new arrangement of length i and annoyance A. So, summing up the total number of ways to get queues of length $i - 1$ and annoyance $A - j$ for $j \in [0..i - 1]$ will give us the total number of ways to get a queue of length i and annoyance A. That is,

$$\text{opt}(i, A) = \sum_{j=0}^{i-1} \text{opt}(i - 1, A - j)$$

However, we can simplify this. With some rearrangement:

$$\text{opt}(i, A) = \left[\sum_{j=1}^{i} \text{opt}(i-1, A-j)\right] + \text{opt}(i-1, A) - \text{opt}(i-1, A-i)$$

$$\text{opt}(i, A) = \left[\sum_{j=0}^{i-1} \text{opt}(i-1, A-(j+1))\right] + \text{opt}(i-1, A) - \text{opt}(i-1, A-i)$$

$$\text{opt}(i, A) = \left[\sum_{j=0}^{i-1} \text{opt}(i-1, (A-1)-j)\right] + \text{opt}(i-1, A) - \text{opt}(i-1, A-i)$$

$$\text{opt}(i, A) = \text{opt}(i, A-1) + \text{opt}(i-1, A) - \text{opt}(i-1, A-i)$$

Order of computation: With our simplified recurrance, each subproblem essentially relies on all the subproblems with queue length $(i-1)$ and all with the same length but less annoyance being solved already. So, we can approach the subproblems in increasing order of A then increasing order of i.

Final answer: The final answer will be $\text{opt}(n, k)$, the number of arrangements with annoyance k and all n people.

Time complexity: There are k subproblems to solve for every i giving $O(nk)$ subproblems and each subproblem takes $O(1)$ time to compute as it simply accesses three previous subproblem solutions.
Since we are not actually sorting the hungers, simply assuming they are sorted, we do not need to add the time complexity of sorting the hungers.
Therefore, the total time complexity is $O(nk)$.

3.1 This sub-question was generally well done by those who seemed to understand their recurrence.

- A surprising number of students incorrectly analyse the complexity of a solution which calculates a sum of n elements per sub-problem to be $O(n * k)$ time. Such responses generally did not get over 9 marks for the incorrect complexity calculations. Exceptions are where there was merit to the calculation.
- Many students simply provided a recurrence, often incorrect, attached with no explanation of why it is correct, nor explanation of how it was derived. Such responses generally did not attain more than 5 marks. There were many different valid ways to approach this, SOME of which were:
  - To define a sub-problem/state-space which considers a queue of length i comprising of the hungriest/least hungry i people. This generally allowed any reasonably formed recurrence to consider what occurs when adding someone into a queue such that they are the person of maximal/minimal hunger in the resulting queue. From there, many ways exist to relate the index of the most recently inserted person to the marginal difference in annoyance in the queue after the insertion. From there, a recurrence can be derived.
  - To define a sub-problem/state-space which considers the first i elements of a newly defined array A[n], such that A[i] represents the distance of the i-th hungriest/least hungry person from their position in the sorted queue in any permutation that results in some annoyance j. One can then relate the possible choices of A[i] to the total annoyance resulting from it. Finally then showing a bijection between all valid A's to make annoyance k and all possible permutations to make annoyance k.
- Many students fail to consider the bounds correctly on the summation. Opt(i, j) = $\sum$ Opt(i - 1, j - k).
  Most students correctly identify that k should start at 0. It should be upper bound at (i - 1) as that is the total number of different places you can insert a person into a queue of length (i - 1). Many instead upper bound it at j or equivalent. This could look like writing an expansion Opt(i - 1, 0) + .. Opt(i - 1, j) which effectively bounds k by j.
- A related issue is that many consider the previous k to be correctly bound at (i - 1) but do not construct base cases compatible with such a recurrence. Clearly if (i - 1) greater than j, one queries opt(x, y) : y less than 0 which therefore, needs to be defined ass 0 for the recurrence to work.

3.2 This question also seemed generally well done by those who understood their recurrence. Most feedback involve arithmetic/algebraic mistakes in derivations that led to slightly incorrect summations. Only one was common across many submissions.
Some students fail to consider cases where opt(i, j) where i greater than j so then the recurrence/cumulative sum update must at some point contain the term -(opt(i - 1, j - (i - 1)). This represents one not considering permutations where person i is effectively inserted outside the bounds of the array. Missing this term was fine for all cases k less than n as the missing term is always 0 for all calculated values but when n greater than k, led to incorrect values.

Question 4

4.1

- Overall, generally well done. Most students got full marks for this part.
- There is a solution for $n = D = 2$ for all three parts of this question – some answers were unnecessarily complicated, with n and d being as large as 8 in some answers.
- Several students failed to handle ties properly - both greedy A and greedy B have a specific way they handle ties, and this affected some students' answers.
- Several students didn't calculate the final wealth of each example, or didn't provide the itinerary for each method but gave the wealth. Be sure to provide everything asked for in the question.
- Some students ignored the directive to provide explicit P and T and argued invalidity of the methods in general – this was not the question.

4.2

- Generally, students completed this question well, though there were a few common mistakes.
- Even for DP problems, you are expected to justify correctness. For most DP problems, this just amounts to explaining where the recurrence comes from.
- Don't forget to check you've properly answered the question – many students did not provide a way to return the itinerary after finding the maximum wealth.
- Several students implemented greedy C as their answer to this question, after proving it doesn't work.
- Subproblems and recurrences need to be completely defined. Specify exactly what the problems are, and what the notation means.