This document gives model solutions to the assignment problems. Note that alternative solutions may exist.

Question 1

You have n identical flowers, which you would like to place into k vases, according to the following rules:

- Each flower must be placed in a vase.
- The ith vase (where $1 \leq i \leq k$) must contain at least 1 and at most f_i flowers.
- No two vases are allowed to contain the same number of flowers.

Your goal is to assign a number of flowers to each vase following the above rules, or to determine that no such assignment exists.

1.1 **[10 marks]** Suppose that $f_1 = \ldots = f_k = n$. Design an algorithm which runs in $O(k)$ time and achieves the goal.

In this case, the capacity requirement is redundant, so we can put any positive number of flowers into each vase.

If $n \geq k(k+1)/2$, we can place i flowers in vase i for all $i \neq k$, and $n-k(k+1)/2+k$ flowers in vase k. By the assumption, $n-k(k+1)/2+k \geq k$, and so all vases have a distinct number of flowers as required. Also this places $(n-k(k+1)/2+k)+\sum_{i=1}^{k-1} i = n-k(k+1)/2+\sum_{i=1}^{k} i = n$ flowers in total (where the last equality follows from the formula for triangular numbers), as required. Producing the assignment of flowers to vases takes $O(k)$ time.

Otherwise, if $n < k(k+1)/2$, no such assignment exists, which we now show. The claim is equivalent to saying that the minimum achievable number of total flowers is $n = k(k+1)/2$. Consider any valid assignment, and let the number of flowers in vase i be c_i. Without loss of generality (since the vases are indistinguishable), suppose the values of c_i are non-decreasing as i increases. If $c_1 > 1$, we could remove one flower from it to achieve another valid assignment with less flowers. Otherwise, consider the first vase which does not contain exactly 1 more flower than the previous vase, supposing one exists. We could remove one flower from this vase, which does not violate distinctness since no vase contains one less flower than the vase we are removing from. Thus we can always decrease the number of flowers in any valid assignment to reach another valid assignment, unless $c_1 = 1$ and each vase contains one more than the previous vase, in which case the total number of flowers is $1+2+\ldots+k = k(k+1)/2$. This shows that $n = 1+2+\ldots+k = k(k+1)/2$ the minimum number of flowers for which a valid assignment exists.

1.2 **[10 marks]** Now we return to the original problem, in which the f_i can take any positive integer values. Design an algorithm which runs in $O(k \log k)$ time and achieves the goal.

Sort the pairs (f_i, i) in non-decreasing order by their first element using merge sort. We will now refer to the vases and the values of f_i by their new indices in this sorted list, since at the end we can convert any index back to its original index simply by taking the second element of the pair.

First, we claim that if any assignment exists, then an assignment exists where the number of flowers in each vase is increasing as we move from left to right. In any assignment which violates this condition, consider the first index i such that the number of flowers in vase i is greater than the number of flowers in vase $i+1$ (which must exist if f is not increasing and

has distinct values). Swapping the number of flowers in vases i and $i+1$ will also result in a valid assignment, since distinctness is preserved, and the number of flowers in vase i will decrease so remain less than f_i, and the number of flowers in vase i will be at most f_i which is at most f_{i+1} as required. Since performing these swaps is equivalent to performing bubble sort, it will eventually result in an assignment where the number of flowers in each vase is increasing.

Consider a strategy which ignores n and places as few flowers as possible, obeying the other restrictions. Since we only need to consider strategies which place an increasing number of flowers in each vase as we move from left to right, to do this, we can sweep from left to right, and place i flowers in the ith vase, since by the argument from 1.1, if we ever did not do this we could decrease the total number of flowers we place. If this is impossible, because $f_i < i$ for some i, then we have determined that no such assignment exists. Also, if $n < k(k+1)/2$, we can report that no assignment exists as we cannot place $k(k+1)/2$ is the minimum possible number flowers we can place.

Now let temporarily allocate i flowers to each vase. We will now perform a procedure updating these allocations, namely, we will sweep from right to left and add as many flowers as possible to each vase, such that

- We do not exceed the capacity of the current vase.
- We place fewer flowers in the current vase than the vase to its right (if one exists).
- We do not place more than n flowers overall. (To do this, we keep track of the number of flowers placed so far.)

Now, if the third constraint never limits us, then we will have placed as many flowers as possible, since the first constraint is strictly necessary, and since from earlier, any assignment violating the second constraint can be transformed into one which conforms to it. If we have placed as many flowers as possible but the number of flowers placed is less than n, we can report that no valid assignment exists. Otherwise, the above procedure will terminate with a valid assignment of flowers to vases, since we will have placed exactly n flowers (due to being constrained by the third constraint, or because n is the minimum possible achievable number of flowers), and it is clear that at every point during the procedure, the assignment will be conform to the capacity and distinctness rules. So we return this final assignment.

Question 2

There are n towns in a line. You are a traveller starting at town 1 and wanting to reach town n. It takes a week to travel from any town to the next.

In each town, there is a market where you can buy rations. In the market of town i, you can buy a week's worth of rations for c_i dollars. Furthermore, you can buy several weeks' rations for later consumption.

Your goal is to calculate the minimum cost to travel from town 1 to town n.

2.1 **[3 marks]** Which town's rations do you want to consume as you travel from town $n-1$ to town n? Provide reasoning to support your answer.

We should consume the rations of town i, where $c_i = \min(c_1, c_2, \ldots, c_{n-1})$. This is because if we consumed the rations of any town amongst towns $1, 2, \ldots, n-1$ which sold more expensive ration, we could have not bought that ration and instead bought one more from town i, decreasing the total cost we spend.

2.2 **[3 marks]** When is the last time (if any) that you should change using from one town's rations to another? Provide reasoning to support your answer.

The last time we will change is when we reach a town which sells rations for the lowest price, that is, at a town i where $c_i = \min(c_1, c_2, \ldots, c_{n-1})$. This is because if we did not switch at such a town, we would be using a ration which costs more than c_i dollars, which we could have not bought and instead bought one from town i to reduce the total cost spent.

2.3 **[14 marks]** Design an algorithm which runs in $O(n)$ time and achieves the goal.

In any optimal solution, to move on from any town i we will use the rations from a town j such that $c_j = \min(c_1, c_2, \ldots, c_i)$, since the rations from towns $1, 2, \ldots, i$ are the only ones we could have up to this point, and if we had used a ration when leaving town i which was not the cheapest out of the first i towns, we could not have bought it and instead bought one of the cheapest rations from the first i towns, reducing the total cost we spend.

Thus the answer is

$$\sum_{i=1}^{n-1} \min(c_1, c_2, \ldots, c_i).$$

To calculate this in $O(n)$ time, we go through the list of c_i values in order, we keep track of min_so_far, initially infinity, and $cost\_so\_far$, initially zero, and every time we encounter a value c_i, we update min_so_far to $\min(c_i, min\_so\_far)$, then increment $cost\_so\_far$ by min_so_far. At the end we return $cost\_so\_far$.

Question 3

You are given a simple directed graph with n vertices and m edges. Each vertex v has an associated 'starting cost' s_v and 'finishing cost' f_v, and each edge e has a corresponding weight w_e. All values w_e, s_v and f_v are positive integers.

For this problem, we define the *score* of a path from vertex a to vertex b as the sum of its starting cost s_a, all edge weights on the path, and its finishing cost f_b. Your goal is to determine the smallest score of any path in the graph.

A path may consist of zero or more edges. The endpoints of a path do not need to be distinct.

3.1 **[8 marks]** Design an algorithm which runs in $O((n+m)\log(n+m))$ time and achieves the goal.

An algorithm running in $\Theta((n+m)^2\log(n+m))$ time will be eligible for up to 6 marks.

Let the original graph be $G = (V, E)$. We construct a new graph G^* with vertex set $V \cup \{s, t\}$, where s is the source and t is the sink. Then:

- for each vertex $v \in V$, we place an edge from s to v with weight s_v,
- for each vertex $v \in V$, we place an edge from v to t with weight f_v, and
- for each edge $e = (u, v) \in E$, we place an edge from u to v with weight w_e.

We now claim that paths in G correspond exactly to s-t paths in G^* with equal score/weight, where:

- starting at vertex a for a cost of s_a corresponds to starting at s and taking the edge from s to a with weight s_a,
- taking regular edges corresponds to taking the identical corresponding edge in the new graph, and
- ending at vertex b for a cost of f_b corresponds to taking the edge from b to t with weight s_b finishing at b.

The new graph G^* has $n+2$ vertices and $2n+m$ edges, so the shortest s-t path in G^* can be computed in $O((n+m)\log n)$ time using Dijkstra's algorithm with an augmented heap. The total weight of this path is the minimum score of a path in G, as required.

3.2 **[6 marks]** Now consider only paths consisting of *at least one edge*.

Design an algorithm which runs in $O((n+m)\log(n+m))$ time and achieves the goal.

We construct a new graph G^{**} with a source vertex s, a sink vertex t and, for each $v \in v$, two corresponding vertices v and v'. Then:

- for each vertex $v \in V$, we place an edge from s to v with weight s_v,
- for each vertex $v \in V$, we place an edge from v' to t with weight f_v,
- for each edge $e = (u, v) \in E$, we place an edge from u to v with weight w_e, and
- for each vertex $v \in V$, we place an edge from v' to v with weight 0.

We now claim that paths in G with at least one edge correspond exactly to s-t paths in G^{**} with equal score/weight. If the original path in G is $v_1v_2v_3\ldots v_{k-1}v_k$, the corresponding path in G^{**} is $sv_1v_2'v_2v_3'v_3\ldots v_{k-1}'v_{k-1}v_k't$, noting that:

- the correspondence between the first edge weight and the cost of the starting vertex is identical to that in 3.1, and likewise for the last edge weight and the cost of the finishing vertex, and
- edges from v_i' to v_i have no weight.

The new graph G^{**} has $2n + 2$ vertices and $3n + m$ edges, so the shortest s-t path in the new graph can be computed in $O((n + m) \log n)$ time using Dijkstra's algorithm with an augmented heap. The total weight of this path is the minimum score of a path in G with at least one edge, as required.

3.3 [6 marks] Now consider only paths consisting of *at least three edges*.

Design an algorithm which runs in $O((n + m) \log(n + m))$ time and achieves the goal.

First, we run Dijkstra's algorithm (with an augmented heap) in G^{**} to find the shortest path from s to not just t, but every vertex u. Record these path weights in an array A.

Next, we construct the reverse graph G^{**}_{rev}, and run Dijkstra's algorithm (with an augmented heap) to find the shortest path from t to every vertex v', and again record the path weights in an array B.

Now, consider an edge $e = (u, v) \in E$. Suppose we have a path p of at least three edges in G, in which e appears but not as the first or last edge. We can write $p = p_1 e p_2$, where p_1 is a path of at least one edge from any vertex to u, and p_2 is a path of at least one edge from v to any vertex. The minimum weights of these paths are exactly the values $A[u]$ and $B[v]$ calculated earlier, so the minimum score of such a path p is $A[u] + w_e + B[v]$. The overall answer is therefound found by taking the minimum value of this expression over all edges $e \in E$.

As shown in 3.2, both executions of Dijkstra's algorithm take $O((n+m) \log n)$ time. We then consider each edge in constant time, allowing us to compute the answer in $O(m)$ time. The first step dominates, so the total time complexity is $O((n + m) \log n)$.

Question 4

You are given an array A, which contains each integer from 1 to n exactly once.

On each move, you can swap the value at index i with the value at index j, for a cost of $|i - j|$ dollars. Your goal is to sort the array, spending as few dollars as possible.

Let $S = \sum_{i=1}^{n} \frac{|A[i] - i|}{2}$.

4.1 **[2 marks]** Show that S is an integer.

Observe that $|x| \equiv x \pmod 2$, so

$$\begin{aligned} 2S &= \sum_{1 \le i \le n} |A[i] - i| \\ &\equiv \sum_{1 \le i \le n} (A[i] - i) \pmod 2 \\ &= \sum_{1 \le i \le n} A[i] - \sum_{1 \le i \le n} i \\ &= 0, \end{aligned}$$

where the last equality follows from the fact that A is a permutation of the integers from 1 to n. Thus $2S$ is even, so S is an integer.

4.2 **[2 marks]** Show that any sequence of swaps which sorts the array will cost at least S dollars.

For each value x in the array, associate it with it an additional value v_x, initially zero. During any sequence of swaps, whenever the value x moves, say from position i to position j, we increment v_x by $|i - j|$.

After any sequence of swaps, $\sum_{x=1}^{n} v_x$ is twice the total cost we have spent, since the swap cost $|i-j|$ is counted twice for each swap (once for each of the values swapped). Also, once the array is sorted, $v_{A[i]}$, the total distance moved by the value $A[i]$, must be at least $|A[i] - i|$, which is the distance between $A[i]$'s starting and ending position. Therefore $\sum_{x=1}^{n} v_x \ge \sum_{i=1}^{n} |A[i] - i| = 2S$.

Thus to sort the array, twice the total cost must be at least $2S$, in other words, any sequence of swaps which sorts the array costs at least S dollars.

4.3 **[8 marks]** Show that the array can be sorted for a total cost of S dollars.

Given any array B with a corresponding S value of $S_B \neq 0$, we will show that we can perform a swap for a positive cost x such that the new array has a corresponding S value of $S_B - x$. By starting with the original array A and repeatedly performing such a swap, we will keep decreasing the value of S until it is 0, meaning that at the end the array is sorted since each index equals the value at that index, and the total amount spent will be the amount S has decreased, which is S_A as required.

It remains to prove that given any array B with $S_B \neq 0$, we can find a swap which will cost the amount which S will decrease by. To see this, for the element at index i, call it 'rightward' if $B[i] > i$ (since $B[i]$ must be moved rightwards to be in the correct position), 'leftward' if

$B[i] < i$ (for the opposite reason), and 'correct' if $B[i] = i$. Consider the array formed by removing all the 'correct' elements, which will be non-empty since not all elements are in the correct position yet. In this remaining array, the first element will be 'rightward' (since if it were to be 'leftward', some element to the left of it would not be 'correct'), and the last element will be 'leftward' (for the opposite reason). Thus in the remaining array there exists a pair of adjacent elements such that the left element is 'rightward' and the right element is 'leftward'.

Swapping these two elements will move each element towards, but not past, its correct position, since if it moved an element past its correct position then some element between the two elements would also not be 'correct'. Thus, with this swap, S (which is half the total distance all elements need to move) will decrease by $|i - j|$ (which is half the total distance the two elements moved towards their correct position), which is the cost of the swap as required.

4.4 **[8 marks]** Design an algorithm which runs in $O(n + S)$ time and finds a list of swaps (each described as a pair of indices) which sorts the array for the minimum possible total cost.

We maintain a doubly linked list of the current state of the array, ignoring all values which are already in the correct position. Each node stores an index in the array (which will increase as the linked list is traversed in the forward direction), and the value at that index. We denote a node by (i, v), where i is the index and v is the value.

We also keep track of references to pairs of nodes in the linked list which we want to swap, using a queue (or a stack). Initially, we traverse the linked list once, and for each adjacent pair of nodes (i_1, v_1) and (i_2, v_2), where $i_1 < i_2$, we push this pair of nodes to the queue if $v_1 > i_1$ and $v_2 < i_2$, that is, if the left value needs to move right, and the right value needs to move left. This initialisation takes $O(n)$.

We now repeat the following process until the linked list is empty: pop a reference to a pair of nodes, say (i_1, v_1) and (i_2, v_2), off the stack, record that we are performing the swap of indices i_1 and i_2, swap the values v_1 and v_2 in the linked list, and update the data structures.

Updating the data structures involves removing any nodes in the right place, that is, any nodes (i, v) where $i = v$, and we only have to check the two nodes which changed. This can be done in $O(1)$ in a doubly linked list. We also have to push to the queue any new pairs of nodes which now are ready to be swapped, and we only need to check the pair (i_1, v_2) and the node to its left (if one exists), and the pair (i_2, v_1) and the node to its right (if one exists), in the same manner as we did at the beginning, pushing a pair if the left value needs to move right and the right value needs to move left. This check also occurs in $O(1)$.

Note that a node can appear at most once in the queue (as its value cannot want to move both left and right), and so performing a swap never invalidates any other pair in the queue, meaning that the queue always contains precisely the nodes we can currently swap.

Thus we can always find and perform a move if one exists in $O(1)$, and at most S moves are used overall since each move spends at least one dollar and we spend S dollars. This gives a final time complexity of $O(n + S)$.