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

Question 9

In your state, there are n towns and m proposed roads. Each road would join a specified pair of distinct towns in both directions, and you have already calculated the cost to the state government of building each road. No two proposed roads join the same pair of towns. You would like to build some of these roads in order to connect the towns, i.e. to allow travel by road between every pair of towns. However, the state budget only allows you to spend C dollars on building roads.

9.1 **[7 marks]** Design an algorithm which runs in $O(m \log n)$ time and determines whether it is possible to connect the towns without exceeding the state budget.

The input consists of the positive integers n, m and C, as well as a list of m triplets of integers of the form (u, v, w), where $1 \leq u \leq v \leq n$, $u \neq v$ and $w > 0$, indicating that there is a proposed road between town u and town v which would cost w dollars for the state government to build.

The output is a boolean value, either "True" or "False", indicating whether it is possible to connect the towns without exceeding the state budget.

For example, if $n = 4$, $m = 4$, $C = 5$ and the triplets are $(1, 2, 2), (2, 3, 2), (3, 4, 2)$ and $(1, 3, 1)$, the correct output is "True" since the roads represented by the 2nd, 3rd and 4th triplets could be built to connect the towns for a total cost of $2 + 2 + 1 \leq C$.

Create an undirected weighted graph where vertices represent towns, edges represent proposed roads, and edge weights represent road costs. Run Kruskal's algorithm using the union-find data structure to find the total weight of the minimum spanning tree. We report "True" if the weight of the MST is at most C, or "False" otherwise (including cases where no MST exists).

Any selection of roads which connects all towns forms a spanning subgraph. The least weight spanning subgraph is the MST, so our selection achieves the minimum cost to connect the towns.

The time complexity of this algorithm is $O(m \log n)$, since the graph has n vertices and m edges.

9.2 **[13 marks]** Having realised that the original task may not be possible, you applied for funding from the federal government. They have agreed to pay for some of the roads on your behalf, so these roads will have no cost to the state budget. However, you will have to fill out a lot of paperwork for each of these roads, so you decide to ask for federal funding for as few roads as possible.

Design an algorithm which runs in $O(m \log n)$ time and determines the smallest number of federally funded roads required in order to connect the towns without exceeding the state budget.

The input format is the same as that in part (a).

The output is an integer, the smallest number of federally funded roads required in order to connect the towns without exceeding the state budget. If it is impossible to connect the towns regardless of how many roads are federally funded, output -1 instead.

For example, if $n = 4$, $m = 4$, $C = 5$ and the triplets are $(1, 2, 2), (2, 3, 2)$ and $(3, 4, 2)$, the correct output is 1 since the roads represented by the 1st and 2nd triplets could be built by the state government for a total cost of $2 + 2 \leq C$, and the road represented by the 3rd triplet could be built by the federal government, resulting in the towns being connected.

The answer is -1 if and only if the graph is not connected. We can test this in $O(n+m)$ using DFS.

A spanning forest with k components (i.e. $n-k$ edges) can be made connected with an additional $k-1$ federal roads. Therefore we want to build a spanning forest with as many edges as possible without exceeding the state budget. At every step of Kruskal's algorithm, the number of edges is increased by one, and the total weight of the forest is minimal among all spanning forests with the same number of edges. Therefore, we run Kruskal's algorithm but terminate early instead of adding any edge that would cause us to exceed the budget, and the answer is the number of remaining components less one, or equivalently $n-1$ less the number of edges taken.

The correctness of this algorithm relies on the fact that Kruskal's algorithm processes the edges in increasing order of weight, so if a spanning tree exists, we will select its cheapest $n-k$ edges, and get the most expensive k for free. It can be proven by induction from $k=n-1$ to $k=1$ that Kruskal's algorithm maintains the minimum spanning forest among those with k components.

Once again, the time complexity is $O(m \log n)$ with the use of the union-find data structure.

Question 10

In your country, n foreign languages are taught in m schools. Each school teaches some (possibly all) of these languages, and all students in the school learn all languages which are offered there.

A conference has been arranged for students from different schools to meet each other. Each school has sent a delegation, with the ith school sending a_i students. One room has been assigned to each language, with the jth language having a room of capacity b_j.

As the organiser of the conference, you must allocate students to rooms according to the following rules:

- each student must be allocated to exactly one room, corresponding to a language taught at their school, and
- no two students from a single school can be allocated to the same room.

10.1 **[20 marks]** Design an algorithm which runs in $O(m^2n^2)$ time and determines whether such an allocation is possible.

The input consists of the positive integers n and m, the positive integers a_i for each $1 \leq i \leq m$, the positive integers b_j for each $1 \leq j \leq n$, as well as a list of distinct pairs of integers of the form (i, j), where $1 \leq i \leq m$ and $1 \leq j \leq n$, indicating that the ith school teaches the jth language.

The output is a boolean value, either "True" or "False", indicating whether a valid allocation is possible.

For example, if $n = 2$, $m = 3$, $a_1 = 2, a_2 = 1, b_1 = 1, b_2 = 1, b_3 = 5$, and the pairs of integers are $(1, 1)$, $(1, 2)$, $(2, 2)$ and $(2, 3)$, the correct output is "True" since the students from the 1st school could go to the rooms with the 1st and 2nd languages, and the student from the 2nd school could go to the room with the 3rd language.

Create a flow network with:

- source s and sink t
- a vertex x_i for each school
- a vertex y_j for each language
- for each school i, an edge (s, x_i) with capacity a_i
- for each language j, an edge (y_j, t) with capacity b_j
- for each language j taught at school i, an edge (x_i, y_j) with capacity 1.

We will interpret a flow in this graph as assigning students to rooms. In particular, an edge (x_i, y_j) carrying one unit of flow will be understood as assigning one student from school i to the room for language j. This ensures that no two students from a single school are allocated to the same room. Furthermore, applying flow conservation at each school vertex x_i, we see that the number of students assigned from a school is at most the size of their delegation a_i, and similarly applying flow conservation at each language vertex y_j, we see that the number of students assigned to a room is at most the capacity of the room b_j. Therefore, any flow in this graph assigns students to rooms without conflicts.

However, we also have to ensure that every student is assigned to a room. This is achieved if and only if the total flow through the cut $(\{s, a_1, \ldots, a_m\}, \{b_1, \ldots, b_n, t\})$, and hence the overall value of the flow, equals $\sum_{i=1}^{m} a_i$, which we will denote by C. Note that the maximum flow cannot exceed C, as this is the total capacity of all edges from s. Therefore we run the

Edmonds-Karp algorithm to find the value of a maximum flow, and report "True" if this value is equal to C or "False" otherwise.

The graph has at most $mn + m + n = O(mn)$ edges. Also, the maximum flow is bounded above by the capacity of the aforementioned cut, which is at most mn. Therefore, the Edmonds-Karp algorithm runs in $O(E|f|) = O(m^2n^2)$ time as required.

Question 11

11.1 **[9 marks]** A 1-indexed array A of n integers is known to be "strongly plateau-shaped", which means that there exist indices l and r such that $1 \leq l \leq r \leq n$, and such that:

- $A[1] < A[2] < \cdots < A[l]$,
- $A[l] = A[l+1] = \cdots = A[r]$, and
- $A[r] > A[r+1] > \cdots > A[n]$.

For example, the following arrays are strongly plateau-shaped:

- $[1, 2, 3, 3, 3, 2, 1]$, with $l = 3$, $r = 5$
- $[1, 5, 7, 4]$, with $l = 3$, $r = 3$
- $[3, 2]$, with $l = 1, r = 1$

However, $[1, 2, 1, 2]$ is not strongly plateau-shaped.

Design an algorithm which runs in $O(\log n)$ time and determines the maximum value which appears in A.

Method I: binary search the sign of the finite difference.

Define a function f, where for $1 \leq i < n$ we define $f(i) = 1$ if $A[i+1] > A[i]$ and $f(i) = 0$ otherwise, and we fix $f(n) = 0$. Now the conditions translate to:

- $f(1), \ldots, f(l-1) = 1$
- $f(l), \ldots, f(n) = 0$

This function is monotonic (in particular, non-decreasing), so we can run a binary search over the range $[1, n]$ to find the smallest index i such that $f(i) = 0$, and return $A[i]$. The constant $f(n) = 0$ guarantees that such an i exists, covering the case where the original array A is strictly increasing.

Each step of the binary search involves a constant time operation, so the time complexity is $O(\log n)$.

Method II: ternary search.

Beginning with the range $[1, n]$, pick two indices $i < j$ which divide the range into thirds. Then:

- if $A[i] < A[j]$, the maximum cannot come from $A[1..(i-1)]$, so we recurse on $A[i..n]$,
- if $A[i] > A[j]$, the maximum cannot come from $A[(j+1)..n]$, so we recurse on $A[1..j]$, and
- if $A[i] = A[j]$, the maximum cannot come from $A[1..(i-1)]$ or $A[(j+1)..n]$, so we recurse on $A[i..j]$.

We continue in this way until the range is of the form $[i, i]$ and return $A[i]$.

Each step of the ternary search reduces the search space by at least one-third, so there are at most $\log_{\frac{3}{2}} n$ such steps. The test at each step takes only constant time, so the time complexity is $O(\log n)$.

11.2 **[11 marks]** A 1-indexed array B of n integers is known to be "weakly plateau-shaped", which means that there exist an index k such that $1 \leq k \leq n$, and such that:

- $B[1] \leq B[2] \leq \cdots \leq B[k]$,
- $B[k] \geq B[k+1] \geq \cdots \geq B[n]$, and
- $B[i] \neq B[j]$, for all indices i and j such that $1 \leq i < k$ and $k \leq j \leq n$.

For example, the following arrays are weakly plateau-shaped:

- $[1, 3, 3, 5, 5, 4, 2]$, with $k = 4$
- $[1, 5, 7, 4]$, with $k = 3$
- $[3, 3, 2]$, with $k = 1$

However, $[1, 2, 4, 4, 3, 3, 2]$ is not weakly plateau-shaped ($k = 3$ is not valid, since $B[2] = B[7]$).

Design an algorithm which runs in in $O(\log^2 n)$ time and determines the maximum value which appears in B.

Suppose that the maximum is known to appear at some index in the range $[l, r)$. Let m be the midpoint of this range. If we can narrow the range to either $[l, m)$ or $[m, r)$ in $O(\log n)$ time (the 'outer' binary search), this will form the basis of a binary search algorithm which solves the problem in $O(\log^2 n)$.

We now run an 'inner' binary search on the range $[l, m]$ to find the smallest index i in this range where $B[i] \geq B[m]$. If this index satisfies $B[i] > B[m]$, then index m must be *after* the peak of the array, so we recurse on $[l, m)$. On the other hand, if $B[i] = B[m]$, then the maximum is not in $B[1..(i-1)]$, nor can it be a value larger than $B[m]$ occurring between indices i and m. Therefore, either the maximum is $B[m]$ itself, or it appears after $B[m]$. In either case, it is safe to narrow our search range to $[m, r)$.

We run the outer binary search on the range $[1, n+1)$, and at each step run the inner binary search in $O(\log n)$. This achieves the target time complexity of $O(\log^2 n)$.

Question 12

There is a shop which has n items, and only one copy of each item. The ith item costs c_i dollars and you will gain v_i happiness points if you buy it.

You have a debit card with an initial positive balance of m dollars, but because of a loophole in the system, you can get away with not having enough money for your final purchase. More precisely, if at some point your card balance is x dollars:

- If $x = 0$, you must not buy any more items.
- If $x > 0$, you can buy the ith item if you have not bought it yet, and your card balance will be updated to $\max(x - c_i, 0)$.

Your goal is to determine the maximum number of happiness points you can get by buying a set of items in some order.

For parts (a) and (b), the input consists of the positive integers n and m, as well as the positive integers c_i and v_i for each $1 \leq i \leq n$.

The output is an integer, the maximum number of happiness points you can get.

For example, if $n = 3, m = 7, c_1 = 3, v_1 = 6, c_2 = 5, v_2 = 5, c_3 = 8$ and $v_3 = 10$, the correct output would be 16, since you could first buy the 1st item then the 3rd item.

12.1 **[13 marks]** Design an algorithm which runs in $O(nm)$ and achieves the goal, under the additional constraint that $c_i = v_i$ for all i.

You may choose to skip part (a), in which case your answer to part (b) will be marked as your answer to part (a) also.

Claim: We will always buy the most expensive item.

Proof: In any sequence of buying items which omits the most expensive item, we can swap the last item bought (the 'free' item) for the most expensive and get more happiness.

Claim: The other items bought can cost up to $m - 1$ dollars.

Proof: If they cost m or more then we don't have enough money to buy these and the most expensive item in any order. But if they cost any less than m, then we can buy them and finish with the best item as desired.

Therefore, we reserve the best item (without loss of generality, suppose this is item n) and find the best combination of the remaining items by dynamic programming, as in Balanced Partition.

Subproblems: for $0 \leq i < m$ and $0 \leq k < n$, let $P(i, k)$ be the problem of determining $\text{opt}(i, k)$, the maximum happiness that can be obtained using up to i cost and only the first k items.

Recurrence: for $i > 0$ and $1 \leq k < n$,

$$\text{opt}(i, k) = \max(\text{opt}(i, k - 1), \text{opt}(i - c_k, k - 1) + v_k),$$

where the cases account for omitting or taking item k respectively.

Base cases: if $i = 0$ (no money to spend) or $k = 0$ (no items to select from), then $\text{opt}(i, k) = 0$.

Final answer: we can spend up to $m - 1$ dollars on all but the best item and then pick that best item, so the answer is

$$\text{opt}(m - 1, n - 1) + v_n.$$

Order of computation: $P(i,k)$ depends on $P(i,k-1)$ and $P(i-c_k,k-1)$, so we can solve subproblems in increasing order of k (and any order of i).

Time complexity: There are $O(nm)$ subproblems, each solved in constant time. The final answer can be computed in $O(n)$. Therefore the total time complexity is $O(nm)$ as required.

12.2 **[7 marks]** Design an algorithm which runs in $O(nm)$ and achieves the goal, with no additional constraints.

Claim: If a set of items can be bought in some order, then it can also be bought in increasing order of cost.

Proof: A set of items can be bought in some order as long as we don't run out of money before getting to the last item in the sequence. If we then rearrange the sequence to put the most expensive item last, this property is maintained.

Therefore we can sort all n items by cost, and consider them in this order.

Claim: The best solution is to spend up to $m-1$ dollars on a subset of the cheapest k items (for some $k \geq 0$), then buy whichever of the last $n-k$ items gives the most happiness.

Proof: Suppose we are buying items $a_1, \ldots, a_{s-1}, a_s$ in order of increasing cost. We cannot run out of money before getting to the last item of this sequence (a_s), so the amount spent on items up to the second last in this sequence (a_{s-1}) must be at most $m-1$. If there is an item more expensive than item a_{s-1} which gives more happiness than item a_s, we would buy that instead of item a_s, since the cost cannot be prohibitive. This concludes the proof, with $k = a_{s-1}$.

We can now proceed by dynamic programming, as in 0–1 Knapsack.

Subproblems: for $0 \leq i < m$ and $0 \leq k < n$, let $P(i,k)$ be the problem of determining $\text{opt}(i,k)$, the maximum happiness that can be obtained using up to i cost and only the cheapest k items.

Recurrence: for $i > 0$ and $1 \leq k < n$,

$$\text{opt}(i,k) = \max(\text{opt}(i,k-1), \text{opt}(i-c_k,k-1)+v_k),$$

where the cases account for omitting or taking item k respectively.

Base cases: if $i = 0$ (no money to spend) or $k = 0$ (no items to select from), then $\text{opt}(i,k) = 0$.

Final answer: we can spend up to $m-1$ dollars on the cheapest k items and then pick the best of the last $n-k$, so the answer is

$$\max_{0 \leq k < n} \left(\text{opt}(m-1,k) + \max_{k < j \leq n} v_j \right).$$

Order of computation: $P(i,k)$ depends on $P(i,k-1)$ and $P(i-c_k,k-1)$, so we can solve subproblems in increasing order of k (and any order of i).

Time complexity: Sorting the items by cost takes $O(nm)$ time using counting sort, since any costs greater than m can be made equal to m without changing the problem. There are $O(nm)$ subproblems, each solved in constant time. The final answer can be computed in $O(n)$ by precomputing

$$f(k) = \max_{k < j \leq n} v_j$$

in decreasing order of k, using the recurrence $f(k) = \max(v_{k+1}, f(k+1))$. Therefore the total time complexity is $O(nm)$ as required.