Question 1. Greedy [15 MARKS]

There are n boxes arranged in a row. From left to right, the lengths of the boxes are $\ell_1, \ldots, \ell_n$. You want to tie up the boxes using strings. You have an unlimited supply of strings, but one string can only be used to tie up a *contiguous block* of boxes with total length is at most L. You want to compute an optimal solution that uses fewest possible strings to tie up all the boxes.

Part (a) [3 MARKS]

What is the necessary and sufficient condition for it to be possible to tie up all the boxes (using as many strings as needed)? No justification is needed.

Each box has length $\leq L$

Part (b) [2 MARKS]

What is the minimum and maximum number of strings that might ever be needed? No justification is needed.

Minimum: 1, Maximum: n

Part (c) [5 MARKS]

Design an $O(n)$ time greedy algorithm for computing a solution that uses the fewest possible strings. Argue why your algorithm runs in $O(n)$ time.

The idea is to start from an end, and always fit as many boxes as possible in the next string. In the algorithm below, m is the number of strings used, and $s[i]$ is the start point of string i.

$m = 1; s[1] = 1; T = \ell_1;$ // Number of strings, start points, and total length so far
FOR $k = 2, \ldots, n$:
  IF $T + \ell_k > L$: // Start a new string
    $m = m + 1; s[m] = k; T = s[m];$
  ELSE: // Continue the current string
    $T += \ell_k;$

Part (d) [5 MARKS]

Prove that your greedy algorithm always returns an optimal solution.

Let $s[\cdot]$ denote the start points returned by the greedy algorithm. Let $s^*[\cdot]$ be start points in an optimal solution that matches the greedy solution for as many start points as possible.

Suppose for contradiction that it does not match greedy on all start points. Let i be the smallest index such that they differ in the start point of string i. Since string $i-1$ started at the same point in both solutions, and greedy algorithm packs as many boxes in a string as possible, we must have $s^*[i] \leqslant s[i]$.

Construct another optimal solution in which we extend string $i-1$ so that string i now starts at $s[i]$, and keep the future start points same as before. This is a valid solution because by greedy algorithm, we know a single string covers everything from $s[i-1]$ to $s[i]-1$. Also, this only reduces the total length of boxes that future strings need to cover. Hence, we found an optimal solution that matches greedy for one more start point, which is a contradiction.

Question 2. Dynamic Programming [20 MARKS]

There are n dice of different colours. These are regular dice with numbers $1, \ldots, 6$ printed on their six sides. Given an integer m, you want to calculate the number of ways in which you can roll the dice to get the sum of numbers to be m. For example, with $n = 2$ dice, there are three ways of getting the sum $m = 3$: the two dice can roll $(1, 2)$ or $(2, 1)$. Note that the dice have different colours, so they are unique.

Part (a) [10 MARKS]

Write the Bellman equation for $\text{dice}(n, m)$, which is the number of ways of getting sum m out of n dice. Clearly identify your base cases. Briefly justify why your Bellman equation is correct.

In the base case of $n = 1$ die, we can uniquely generate sums 1 through 6 but no other sums. For $n > 1$, the last die can turn up $t \in \{1, \ldots, 6\}$ (each is counted as one possibility), and in that case, we need $m - t$ sum from the rest $n - 1$ dice.

$$\text{dice}(n, m) = \begin{cases} 0, & \text{if } n = 1 \text{ and } (m \leqslant 0 \vee m > 6), \\ 1, & \text{if } n = 1 \text{ and } 1 \leqslant m \leqslant 6, \\ \sum_{t=1}^{6} \text{dice}(n-1, m-t), & \text{otherwise.} \end{cases}$$

Part (b) [5 MARKS]

Write a bottom-up dynamic program that implements your Bellman equation and computes $\text{dice}(n, m)$ in $O(n \cdot m)$ time.

Initialize $\text{dice}(1 \ldots n, -5 \ldots m)$ to 0;
FOR $a = 1, \ldots, n$:
    FOR $b = 1, \ldots, m$:
        IF $(a == 1 \wedge b < 7) : \text{dice}(a, b) = 1$;
        ELSE IF $(a == 1 \vee b < 0) : \mathit{dice}(a, b) = 0$;
        ELSE:
            $\text{dice}(a, b) = \text{dice}(a-1, b-1) + \text{dice}(a-1, b-2) + \text{dice}(a-1, b-3) + \text{dice}(a-1, b-4) + \text{dice}(a-1, b-5) + \text{dice}(a-1, b-6)$;

Part (c) [5 MARKS]

How would you modify your algorithm from part (b) so that it runs in time $O(n \cdot \min(n, m))$?

IF $m > 6n$: RETURN 0;
ELSE: Run the algorithm of part (b).

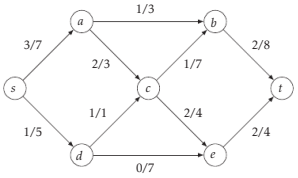
This is justified because it is not possible to generate sum more than $6n$ with n dice. The algorithm takes $O(n \cdot m)$ time as before when $m \leqslant 6n$, but takes $O(1)$ time otherwise. Hence, it takes at most $O(n \cdot \min(n, m))$ time.

Question 3. Network Flow I [15 MARKS]

Consider the following network N and an initial flow f. Each edge is labeled a/b, where b is the capacity of the edge and a is the flow it carries under f.

Part (a) [6 MARKS]

Find a maximum s-t flow in this network N by starting from f, and using augmenting paths in the residual graph as in the Ford-Fulkerson algorithm. Show your work. For each iteration, state the augmenting path you use (including which edges are forward and which are reverse), the amount of additional flow you send along the path, and the total value of the flow at the end of the iteration.



Augmenting path	Additional flow	Total flow
$s \to a \to b \to t$ (all forward)	2	6
$s \to a \to c \to b \to t$ (all forward)	1	7
$s \to d \to e \to t$ (all forward)	2	9
$s \to d \to e \to c \to b \to t$ ($e \to c$ reverse, rest forward)	2	11

Part (b) [3 MARKS]

Based on your maximum flow from (a), find a minimum s-t cut in network N. List the edges that go across your cut from the s side to the t side. No other justification is needed.

As far as I can see, there are two min-cuts. Students are free to identify either of them.
Min-cut 1: $\{a \to b, a \to c, s \to d\}$
Min-cut 2: $\{a \to b, a \to c, d \to c, e \to t\}$.

Part (c) [3 MARKS]

From the edges listed in part (b) that go across the min s-t cut, find an edge such that increasing its capacity by one would increase the maximum s-t flow. State the edge and find an augmenting path in the residual graph of your flow from part (a) after increasing the capacity of this edge by one.

Again two possible solutions:
Solution 1: Edge $a \to b$, augmenting path $s \to a \to b \to t$
Solution 1: Edge $a \to c$, augmenting path $s \to a \to c \to b \to t$

Part (d) [3 MARKS]

Give an example of one edge across your cut in part (b) such that increasing its capacity by one would NOT increase the max flow. State the edge, and explain why this is possible despite the max-flow min-cut theorem which states that the max s-t flow in a network is equal to the minimum capacity of any s-t cut.

Both $d \to c$ and $e \to t$ are correct edges for the first part of the question.

Justification: This does not contradict the max-flow min-cut theorem because this edge is part of *one* min-cut. But even if its capacity is increased by 1, there remains another min-cut in the graph with the same capacity as before. That's why the flow value does not increase.

Question 4. Network Flow II [10 MARKS]

There are n families who go to a resort together for a vacation. Family i consists of x_i people. There are m activities available at the resort, and at most y_j people are allowed to participate in activity j. As the trip planner, your job is to assign people from all families to activities. But you need to make sure that each person from each family is assigned to exactly one activity, no activity j is assigned more than y_j people, and no two people from the same family are assigned to the same activity.

By reducing this problem to network flow and assuming you are given access to a polynomial-time algorithm for network flow, design a polynomial-time algorithm that either produces a feasible assignment of people to activities or declares that no such assignment exists.

Part (a) [2 MARKS]

TRUE/FALSE: "*An algorithm is polynomial-time if it runs in time polynomial in n, m, $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$.*" (If this is true, explain why. If this is false, describe the quantities in which the running time should really be polynomial.)

FALSE. It should be polynomial in n, m, $\log x_1, \ldots, \log x_n$, and $\log y_1, \ldots, \log y_m$.

Part (b) [8 MARKS]

Describe your flow network. Clearly state the vertices (including source and target vertices), the edges, and the edge capacities. Explain how computing the maximum flow in this network helps you solve your activity planning problem.

One vertex a_i for each family i and one vertex b_j for each activity j. In addition, one source s and one target t.

Edges:

1. For each family i: edge $s \to a_i$ of capacity x_i.
2. For each activity j: edge $b_j \to t$ of capacity y_j.
3. For each family i to each activity j: edge $a_i \to b_j$ of capacity 1.

The $b_j \to t$ flow constraint ensures that no more than y_j people are placed on activity j. Flow capacity of 1 on $a_i \to b_j$ ensures that no two people from the same family are placed on the same activity. If the maximum flow is equal to $\sum_i x_i$, then we can place all members of all families, and every $a_i \to b_j$ edge that carries a flow of 1 means one (any) member of family i can be placed on activity j. If the maximum flow value is less than $\sum_i x_i$, then there is no feasible assignment.

Question 5. Linear Programming [15 MARKS]

Part (a) [5 MARKS]

A cargo plane has two sections (front and rear), whose weight and volume limits are given in the first table below. These sections can be loaded with two types of cargo, whose volume and profit per unit weight are given in the second table below.

Section	Weight limit (tonnes)	Volume limit (m^3)
Front	5	300
Rear	3	200

Cargo	Volume (m^3/tonne)	Profit (CA$/tonne)
C1	35	15
C2	53	32

Write a linear program that decides how many tonnes of each cargo should be loaded into each section to maximize profit without violating the weight and volume limits of each section. Clearly indicate what each variable of your LP means. Assume that you have unlimited supply of each cargo and that the number of tonnes loaded can be any non-negative real number. (You only need to write the LP; you do not need to actually solve it.)

Variables: $x_{1,1}, x_{1,2}$ denote the weight of cargo 1 that should go to front and rear, and $x_{2,1}, x_{2,2}$ denote the weight of cargo 2 that should go to front and rear.

$$\begin{aligned}
&\text{Maximize } 15\cdot(x_{1,1}+x_{1,2})+32\cdot(x_{2,1}+x_{2,2})\\
&\text{s.t.}\\
&x_{1,1}+x_{2,1} \leqslant 5\\
&x_{1,2}+x_{2,2} \leqslant 3\\
&35x_{1,1}+53x_{2,1} \leqslant 300\\
&35x_{1,2}+53x_{2,2} \leqslant 200\\
&x_{1,1}, x_{1,2}, x_{2,1}, x_{2,2} \geqslant 0
\end{aligned}$$

Part (b) [5 MARKS]

Consider the following program, which is not linear because one of its constraints involves absolute values. Show that this can be converted to an equivalent linear program by replacing the problematic constraint by one or more linear constraints. Argue why your LP is equivalent (i.e. produces the same optimal solution). Here, $|\cdot|$ is the absolute value function, i.e., $|z| = z$ if $z \geqslant 0$ and $|z| = -z$ if $z < 0$.

$$\begin{aligned}
&\text{Maximize } 2a - 3b\\
&\text{s.t.}\\
&|a-5|+|a+b-3| \leqslant 10\\
&a, b \geqslant 0
\end{aligned}$$

Replace $|a-5|+|a+b-3| \leqslant 10$ with four constraints:
$+(a-5)+(a+b-3) \leqslant 10$
$-(a-5)+(a+b-3) \leqslant 10$
$+(a-5)-(a+b-3) \leqslant 10$
$-(a-5)-(a+b-3) \leqslant 10$
All four left hand sides being less than or equal to 10 is equivalent to their maximum being less than or equal to 10, but their maximum is precisely $|a-5|+|a+b-3|$.

Part (c) [5 MARKS]

Consider the following program, which is not linear because the objective function involves absolute value. Show that it can still be solved in polynomial time by solving one or more LPs.

$$\begin{aligned}
&\text{Maximize } |c^T x|\\
&\text{s.t.}\\
&Ax \leqslant b\\
&x \geqslant 0
\end{aligned}$$

Solve two LPs. One that maximizes $c^T x$ and one that minimizes $c^T x$. Take the maximum of the absolute values of both optimal objective values.

Question 6. Complexity [10 MARKS]

Part (a) [5 MARKS]

Consider the following two problems.

HAM-CYCLE:
Input: An undirected graph $G = (V, E)$.
Question: Does G have a Hamiltonian cycle (i.e. a simple cycle containing all $|V|$ vertices)?

HALF-CYCLE:
Input: An undirected graph $G = (V, E)$.
Question: Does G have a simple cycle containing at least $\lfloor\, |V|/2 \,\rfloor$ vertices?

Show that HALF-CYCLE is NP-complete. For the hardness part, use HAM-CYCLE in your reduction. (You can assume that HAM-CYCLE is known to be NP-complete.)

For membership in NP, we note that given a HALF-CYCLE solution, a TM can verify in polytime that it is a simple cycle and that it contains at least $\lfloor |V|/2 \rfloor$ vertices.

For reduction, let G be an instance of HAM-CYCLE. Construct a graph $G' = (V', E')$ where V' consists of V as well as $n = |V|$ additional isolated vertices. These vertices cannot be part of any cycle (of length at least 2). Hence, G' has a HALF-CYCLE iff G' has a cycle of length at least n iff G has a cycle of length at least n (i.e. a HAM-CYCLE).

Part (b) [5 MARKS]

Consider the following two problems. They are decision versions of the corresponding optimization problems that we saw in class.

MAX-CUT:
Input: An undirected graph $G = (V, E)$ and an integer ℓ.
Question: Is there a partition of V into (A, B) such that at least ℓ edges have one endpoint in A and the other in B?

MAX-2-SAT:
Input: A 2-CNF formula φ and an integer t. (In a 2-CNF formula, each clause has exactly two literals.)
Question: Does there exist a truth assignment under which at least t clauses of φ are satisfied?

Show that MAX-2-SAT is NP-complete. For the hardness part, use MAX-CUT in your reduction. (You can assume that MAX-CUT is known to be NP-complete.)

[Hint: You can construct a 2-CNF formula which has two clauses for every edge in G.]

For membership in NP, we note that given a truth assignment in question, a TM can easily verify in polytime that it is a valid truth assignment and that it satisfies at least t clauses of φ.

For the reduction, let (G, ℓ) be an instance of MAX-CUT. Construct a 2CNF formula φ as follows. Create a variable x_u for each vertex u. For each edge (u, v), create two clauses: $(x_u \vee x_v)$ and $(\bar{x}_u \vee \bar{x}_v)$. Set $t = m + \ell$.

There is a 1-1 correspondence between truth assignments and cuts, which is given by x_u = TRUE iff $u \in A$. For each edge across the cut, both clauses are satisfied. For each edge within A or within B, exactly one of the two clauses is satisfied. So if there are x edges across the cut, then the number of clauses satisfied is $m + x$. Hence, there is a cut of size $\geqslant \ell$ iff there is a truth assignment satisfying

$\geqslant m + \ell$ clauses.

Question 7. Approximation Algorithms [15 MARKS]

Consider the following problem.

d-REGULAR-MAX-INDEPENDENT-SET:
Input: An undirected graph $G = (V, E)$ in which every vertex has degree exactly d.
Output: A maximum cardinality independent set I^* of G (an independent set is a subset of vertices such that no two of them are connected by an edge).

Consider the following greedy algorithm for this problem.

```
GreedyIndSet(G):
    I ← ∅;
    while G has at least one remaining vertex:
        v ← any remaining vertex in G;
        I ← I ∪ {v};
        Delete v and all its neighbours from G;
    return I
```

Part (a) [5 MARKS]

Argue that the greedy algorithm always returns an independent set.

By induction, we can show that no nodes remaining in G are connected to any node in I. This is trivially true initially when $I = \varnothing$. When v is added to I, v and all its neighbours are deleted from G, and thus, this property is maintained. Hence, every time we add a node to I, it is not connected to any previously added nodes, maintaining the fact that I is an independent set.

Part (b) [10 MARKS]

Find the largest c such that the greedy algorithm is guaranteed to return an independent set containing at least $c \cdot |V|$ vertices. Your answer should be in terms of only d. Justify your answer.

The optimal answer is $c = 1/(d + 1)$.

To see that the algorithm always returns a set containing at least $|V|/(d+1)$ vertices, note that for each node v that we add to I, we remove at most $d + 1$ nodes from the graph (v and its at most d neighbours). Hence, the algorithm continues for at least $|V|/(d + 1)$ iterations.

To prove that this is the largest possible, consider the a graph which consists of isolated copies of K_{d+1} (complete graph on $d+1$ vertices). The algorithm always picks one vertex and removes the K_{d+1} copy it was taken from. Hence, the algorithm returns an independent set of size $|V|/(d + 1)$. Hence, no better bounds are possible.

(Bonus) Part (c) [10 MARKS]

Find the largest ρ such that the greedy algorithm is a ρ-approximation. That is, on any graph G, it produces an independent set of cardinality at least $\rho \cdot OPT$, where OPT is the maximum cardinality of any independent set in G. Your answer should be in terms of only d. Justify your answer.

NOTE: This subquestion is for extra credit. Solve this only after attempting all other questions.

The optimal answer is $\rho = 2/(d+1)$.

To see that it does give this approximation, we just need to show that no independent set can be larger than $|V|/2$. This is true because if we have an independent set of size k, its vertices have $k \cdot d$ incident edges. The other endpoints of all these edges must be the remaining $|V| - k$ vertices. But each such vertex has d incident edges. Hence, $k \cdot d \leqslant (|V| - k) \cdot d \Rightarrow k \leqslant |V|/2$.

To show that this is the largest possible, we need to show a graph where there is an independent set of size $|V|/2$ but our algorithm outputs $|V|/(d+1)$. Consider a d-regular bipartite graph with $|V|/2$ vertices on both sides. There is an independent set of size $|V|/2$, but if our algorithm keeps picking from the two sides alternatively, then it will generate value $|V|/(d+1)$.