

Endsem exam

Total Marks: 80

Time: 180 minutes

Instructions.

- Please write your answers clearly, concisely, and to the point.

Question 1 [4 marks]. Suppose a problem X is NP-complete. State TRUE or FALSE for each of the below. 1 mark for correct answer, -0.5 for incorrect answer, -0.5 for writing ambiguously, for example, T. No explanation expected.

- (a) We know for sure that X does not have a polynomial time algorithm.

False.

- (b) We know for sure that X has an exponential time ($O(2^{n^c})$ for input size n and some constant c) algorithm.

True. We know that for yes instances there is polynomial size proof. We can run an algorithm that goes over all possible proofs of polynomial size and verify whether there is an acceptable proof. Here by all possible proofs, we just mean all possible 0-1 strings of polynomial size.

- (c) The complement of X is also in NP (problem Y is called complement of X if for any input α , α is a yes input for X if and only if α is a no input for Y).

False. There need not be a proof for no instances of problem X .

- (d) Designing a polynomial time algorithm for X will imply $P = NP$.

True.

Question 2 [4 marks]. Find the value of $2^{x+218} \bmod 15$, where x is the number formed by last two digits of your roll number. Note that $(a \times b) \bmod c = ((a \bmod c) \times (b \bmod c)) \bmod c$.

Only the answer is expected. One can compute the answer by repeated squaring as follows. Say, $x = 43$. Then $x + 218 = 261$. Write 261 as sum of powers of 2.

$$261 = 2^8 + 2^2 + 2^0.$$

By repeated squaring and taking modulo 15, we get that

- $2^{2^0} \bmod 15 = 2$
- $2^{2^1} \bmod 15 = 4$
- $2^{2^2} \bmod 15 = 1$
- $2^{2^3} \bmod 15 = 1$
- $2^{2^4} \bmod 15 = 1$
- $2^{2^5} \bmod 15 = 1$
- $2^{2^6} \bmod 15 = 1$
- $2^{2^7} \bmod 15 = 1$

- $2^{2^8} \bmod 15 = 1$

Hence,

$$2^{261} \bmod 15 = 2^{2^8+2^2+2^0} \bmod 15 = 1 \times 1 \times 2 = 2$$

The answer for any x is given below.

$x \bmod 4$	$2^{x+2^{18}} \bmod 15$
0	4
1	8
2	1
3	2

Side remark: why do you think the value of 2^y is repeating after every 4 steps. How is 4 related to 15?

Question 3. Given two sorted integer arrays A and B (with all distinct numbers in them) of size n and a number k , we want to find the rank k element in the union of the two arrays. We want to find it by accessing only $O(\log n)$ entries in the two arrays. Let us design a recursive algorithm for this. Assume both arrays are indexed from 0 to $n - 1$. Consider a function $\text{RANKELEMENT}(p, q, r, s, t)$ which is supposed to return the rank t element in the union of two subarrays $A[p \dots q - 1]$ and $B[r \dots s - 1]$. We will get the desired answer by calling $\text{RANKELEMENT}(0, n, 0, n, k)$.

(a) [2+2 marks] Fill in the blanks in the following pseudocode (we are ignoring the base cases, when there is no recursive call).

```

RANKELEMENT( $p, q, r, s, t$ ){
     $midA \leftarrow \lfloor (q - p)/2 \rfloor$ ;
     $midB \leftarrow \lfloor (s - r)/2 \rfloor$ ;
    if  $midA + midB < t$  then
        if  $A[p + midA] < B[r + midB]$  then
            return  $\text{RANKELEMENT}(p + midA, q, r, s, t - midA)$ .
        else
            (similar to above, not required to fill in).
    else
        if  $A[p + midA] < B[r + midB]$  then
            return  $\text{RANKELEMENT}(p, q, r, r + midB, t)$ .
        else
            (similar to above, not required to fill in).
}

```

-0.5 for missing writing $midA$ instead of $p+midA$.

(b) [2 marks] Argue that the time complexity of the above algorithm is $O(\log n)$.

In each recursive call, the size of one of the two arrays gets halved, that is why in $O(\log n)$ rounds, we will get size 1 for both the arrays. Then we will get the answer directly (base case).

Question 4 [8 marks]. Suppose there are n objects with their weights being $w_1, w_2, \dots, w_n \geq 0$ and their values being $v_1, v_2, \dots, v_n \geq 0$. You need to select a **contiguous subset** of the objects such that the total weight is bounded by a given target W , while the total value is maximized. A contiguous subset means a subset of the kind $\{i, i + 1, i + 2, \dots, j\}$ for some $j \geq i$. Design an $O(n \log n)$ time algorithm for this (a better time complexity is also acceptable if correct, but no extra marks). Assume arithmetic operations can be done in constant time.

Approach 1:

First in $O(n)$ time, compute sums of weights of all the subarrays starting from 1. That is, we want to compute $W_i = w_1 + w_2 + \dots + w_i$ for all $1 \leq i \leq n$. We do it as follows.

$W_1 = w_1$.

for $i \leftarrow 2$ to n : $W_i \leftarrow W_{i-1} + w_i$.

Now, for any subset $\{i + 1, i + 2, \dots, j\}$ the sum can be computed as $W_j - W_i$, in constant time.

Now, go over all possible starting index i for the subarray. The ending index for the subarray should be the largest j such that weight of the subarray is bounded by W . We can do a binary search for ending index j : if sum of weights in $W_j - W_{i-1} > W$ then we should go to a smaller j , otherwise larger j .

This way, we get the maximum possible value among all subsets starting from i in $O(\log n)$ time. Doing this for every choice of i and taking maximum will take $O(n \log n)$ time.

Approach 2: There is an $O(n)$ solution for this as well.

- Keep two pointers i and j , starting with $i = j = 1$.
- Maintain S as the sum of weights $S = w_i + w_{i+1} + \dots + w_j$.
- Maintain V as the sum of values $V = v_i + v_{i+1} + \dots + v_j$.
- $\text{MaxValue} \leftarrow 0$.
- for $j \leftarrow 1$ to n
 - Increase i till the first point such that S comes down to at most W .
 - Update $\text{MaxValue} \leftarrow \max(\text{MaxValue}, V)$.

Running time: Both pointers can increase at most $n - 1$ times. Whenever one of the pointer increases, we need to do an update and check, which can be done in constant time. Hence, overall time is $O(n)$.

Correctness (not expected): The algorithm is not considering all subarrays. But, that is not a problem. We increase i only when $w_i + \dots + w_j > W$ for the current value of j . Hence, we don't need to consider the subsets $\{i, i + 1, \dots, j'\}$ for any $j' \geq j$.

Question 5. Recall the census rounding problem from the quiz. We have collected some statistics about populations in different districts and in different age groups. These numbers are recorded in multiples of one lakh and they are written in a matrix. See the below table, for example.

	Age group 1	Age group 2	Age group 3	Sum
District 1	18.77	10.33	5.78	34.88
District 2	15.77	7.00	5.61	28.38
District 3	14.34	8.69	6.71	29.74
Sum	48.88	26.02	18.10	93.00

Assume that the sum of all entries is integral. Each matrix entry, as well as the row sums and column sums need to be rounded up or down. That is, α must be replaced with $\lfloor \alpha \rfloor$ or $\lceil \alpha \rceil$. After the rounding, the sums should remain valid. A good rounding scheme for above example is as follows. The goal is to prove that this is always possible.

	Age group 1	Age group 2	Age group 3	Sum
District 1	18	11	6	35.00
District 2	16	7	5	28.00
District 3	15	8	7	30.00
Sum	49.00	26.00	18.00	93.00

(a) [8 marks] For any given such a matrix, describe a procedure to construct an appropriate flow network (directed graph with integer edge capacities, one source and one sink) and show that

- The matrix entries can be used to define a flow with maximum possible outgoing flow from source.

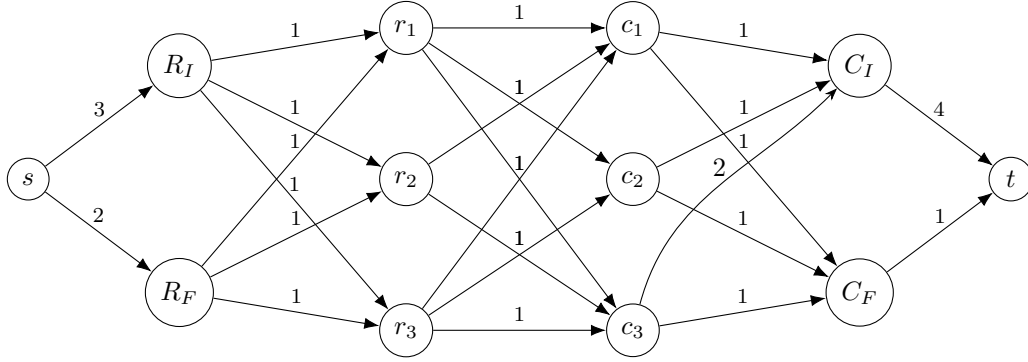


Figure 1: Flow network for the below example.

- Any maximum flow with integral flow for each edge can be used to find an appropriate rounding of the given matrix.

A solution with lower bound on the flow in each edge is not considered correct. The question clearly mentioned that you can only have edge capacities.

As was done in the quiz, we first subtract $\lfloor \alpha \rfloor$ from each matrix entry α (see Figure 2 for a direct construction, without this subtraction). We should adjust the row sums and column sums appropriately. In the given example, we will have

	Age group 1	Age group 2	Age group 3	Sum
District 1	0.77	0.33	0.78	1.88
District 2	0.77	0.00	0.61	1.38
District 3	0.34	0.69	0.71	1.74
Sum	1.88	1.02	2.10	5.00

After this modification, let the i th row sum be rowsum_i and let j th column sum be columnsum_j . Let the sum of all matrix entries be totalsum .

See Figure 1 for the network construction for the above example. We create the following vertices

- One vertex for each row and column, say r_i for row i and c_j for column j .
- A source vertex s .
- A sink vertex t .
- Two vertices R_I and R_F .
- Two vertices C_I and C_F .

The intuition is to split the row sum into integral and fraction parts. Corresponding flows will come from R_I and R_F .

We create the following edges

- Edges (r_i, c_j) with capacity 1 if and only if the (i, j) entry in the matrix is nonzero.
- for each row i , edge (R_I, r_i) with capacity $\lfloor \text{rowsum}_i \rfloor$.
- for each row i , edge (R_F, r_i) with capacity 1, if and only if rowsum_i is not integral.
- Edge (s, R_I) with capacity $\sum_i \lfloor \text{rowsum}_i \rfloor$.
- Edge (s, R_F) with capacity $(\text{totalsum} - \sum_i \lfloor \text{rowsum}_i \rfloor)$.
- for each column j , edge (c_j, C_I) with capacity $\lfloor \text{columnsum}_j \rfloor$.

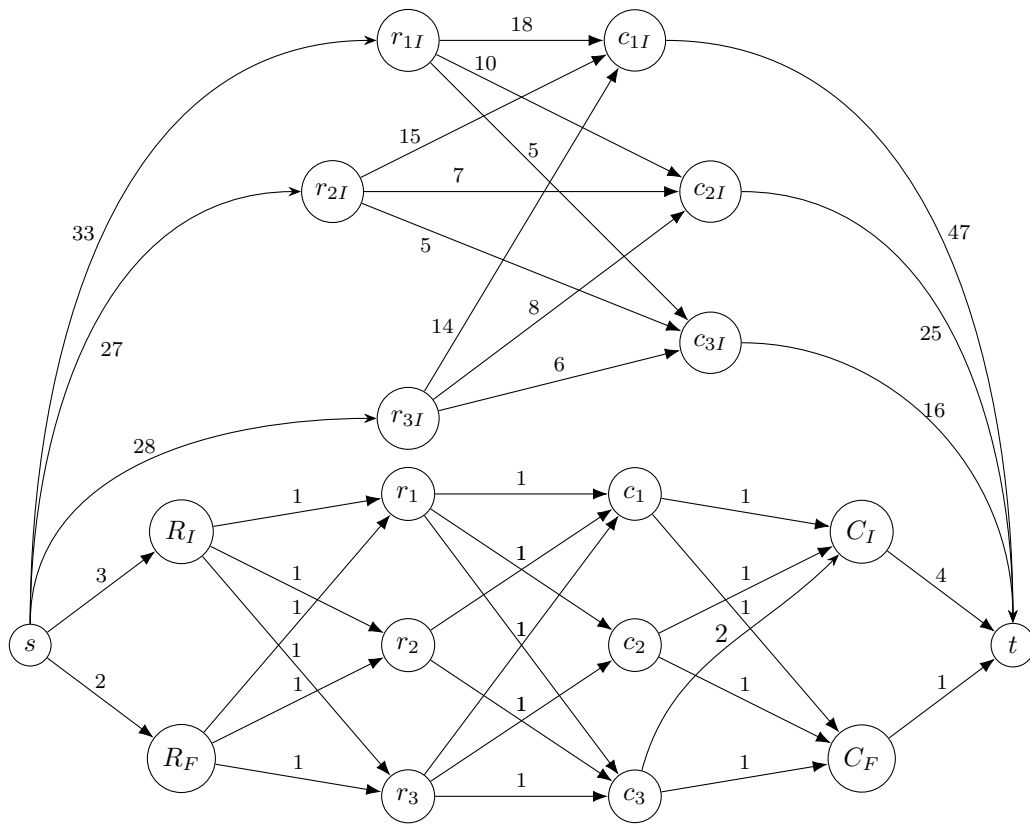


Figure 2: Alternate construction of flow network without subtracting integral part.

- for each column j , edge (c_j, C_F) with capacity 1, if and only if columnsum_j is not integral.
- Edge (C_I, t) with capacity $\sum_j \lfloor \text{columnsum}_j \rfloor$.
- Edge (C_F, t) with capacity $(\text{totalsum} - \sum_j \lfloor \text{columnsum}_j \rfloor)$.

The matrix entries can be used to define a flow with maximum possible outgoing flow from source. The following flow can be defined:

- flow in edge (r_i, c_j) is $x_{i,j}$, the (i, j) entry
- flow in edges (s, R_I) , (s, R_F) , and (R_I, r_i) with full capacity.
- flow in edges (C_I, t) , (C_F, t) , and (c_j, C_I) with full capacity.
- flow in edge (R_F, r_i) is $(\text{rowsum}_i - \lfloor \text{rowsum}_i \rfloor)$.
- flow in edge (c_j, C_F) is $(\text{columnsum}_j - \lfloor \text{columnsum}_j \rfloor)$.

It is straightforward to see that at every vertex, incoming flow is equal to outgoing flow. Also, flow in every edge is at most its capacity. Total outgoing flow from source is

$$\sum_i \lfloor \text{rowsum}_i \rfloor + (\text{totalsum} - \sum_i \lfloor \text{rowsum}_i \rfloor) = \text{totalsum}.$$

This is the maximum possible flow because both the edges from s have flow with full capacity.

Any maximum flow with integral flow for each edge can be used to find an appropriate rounding of the given matrix.

Any maximum flow must have edges (s, R_I) and (s, R_F) with full capacity flow. Since (s, R_I) has a full capacity flow, the same should be true for each edge (R_I, r_i) . Hence, each edge (R_I, r_i) must have a flow of $\lfloor \text{rowsum}_i \rfloor$ units.

Let $f_{i,j}$ be the flow in edge (r_i, c_j) , which will be either 0 or 1. Thus, $\{f_{i,j}\}$ will be a valid rounding for the modified matrix. By flow conservation, the sum of the i -th row will be flow in edge (R_I, r_i) plus flow in edge (R_F, r_i) . Hence, the sum of i -th row will be between $\lfloor \text{rowsum}_i \rfloor$ and $\lfloor \text{rowsum}_i \rfloor + 1$, as desired. The same can be argued for column sums.

Again due to flow conservation, the sum of all row-sums will be equal to maxflow, which is totalsum. Thus, we have an appropriate rounding of the modified matrix.

To get a rounding of the given matrix, we can simply add $\lfloor \alpha_{i,j} \rfloor$ back to the (i, j) entry. The row sums, column sums, and total sum will be appropriately adjusted.

Figure 2 gives an alternate construction that does not need to subtract $\lfloor \alpha_{i,j} \rfloor$.

(b) [4 marks] Consider the more general case, when the total sum of all entries need not be integral. Now, along with matrix entries, row sums, and columns sums, we also want to round the total sum up or down, while all sums remain valid. Prove that this is always possible.

To see this we construct the same network as in part (a) with one change.

- We make the capacity of edge (s, R_F) equal to $(\lceil \text{totalsum} \rceil - \sum_i \lfloor \text{rowsum}_i \rfloor)$.
- We make the capacity of edge (C_F, t) equal to $(\lceil \text{totalsum} \rceil - \sum_j \lfloor \text{columnsum}_j \rfloor)$.

Just like part (a), the matrix entries will give a feasible flow. The total outgoing flow from s will be totalsum. But, recall that when capacities are integral, then the max flow value should be integral. Hence, we can conclude that max flow is at least $\lceil \text{totalsum} \rceil$. But total outgoing capacity from s is the same, $\lceil \text{totalsum} \rceil$, hence, max flow is equal to this quantity. Consider an integral flow with outgoing flow from s being equal to $\lceil \text{totalsum} \rceil$. As argued in part (a), the integral flow give us a valid rounding of the matrix. In particular, the sum of all entries is rounded to $\lceil \text{totalsum} \rceil$.

Question 6 (randomized algorithm) [8 marks]. Consider the following algorithm to compute minimum and second minimum element in an array A .

```

min ← ∞;
second-min ← ∞;
for i → 0 to n - 1 {
    if A[i] < second-min then
        if A[i] < min then
            second-min ← min; min ← A[i];
        else
            second-min ← A[i];
}

```

In worst case, the algorithm makes $2n$ comparisons. Assume that the elements in array A are arranged in a random permutation (each permutation is equally likely). Prove that the expected number of comparisons will be $n + O(\log n)$.

Let X_i be the number of comparisons made in the iteration i . It is a random variable. Total expected number of comparisons will be $X = \sum_{i=0}^{n-1} X_i$. By linearity of expectation,

$$\mathbb{E}[X] = \sum_{i=0}^{n-1} \mathbb{E}[X_i].$$

To compute $\mathbb{E}[X_i]$, first observe that it can take only two values 1 or 2. Let us compute the probability of each possibility. The first comparison is always made. The second comparison is made only if the $A[i] < \text{second-min}$. Equivalently, the second comparison is made only if $A[i]$ is the minimum or second-minimum among $A[0], A[1], \dots, A[i]$.

Fix a set S of $i + 1$ elements. Conditioned on the event that the set of first $i + 1$ elements is S , we can say that each of the $(i + 1)!$ permutations of S are equally likely. Since, each permutation is equally likely, we can say

$$\Pr(A[i] \text{ is the minimum among } \{A[0], A[1], \dots, A[i]\}) = 1/(i + 1).$$

Similarly,

$$\Pr(A[i] \text{ is the second-minimum among } \{A[0], A[1], \dots, A[i]\}) = 1/(i + 1).$$

Since, the two probabilities are independent of the choice of S , the same probabilities hold even when we don't condition on S .

Finally, we can say that

$$\Pr(\text{there are two comparisons in the } i\text{th iteration}) = 2/(i + 1).$$

Coming back to $\mathbb{E}[X_i]$, we can write

$$\mathbb{E}[X_i] = 2 \times \frac{2}{i + 1} + 1 \times \left(1 - \frac{2}{i + 1}\right) = 1 + \frac{2}{i + 1}.$$

Now, we can plug this into the equation for $\mathbb{E}[X]$

$$\mathbb{E}[X] = \sum_{i=0}^{n-1} \mathbb{E}[X_i] = \sum_{i=0}^{n-1} \left(1 + \frac{2}{i + 1}\right) \leq n + (2 + 2 \log n).$$

The last inequality was seen in the class.

Question 7 (approximate Knapsack). Suppose there are n objects with their weights being w_1, w_2, \dots, w_n and their values being v_1, v_2, \dots, v_n (all positive integers). You need to select a subset of the objects such that the total weight is bounded by a given target W , while the total value is maximized. We have discussed ideas that lead to an algorithm with pseudo-polynomial time, that is, the time is proportional to the given numbers (weight/values). We want to design a polynomial time algorithm that may give an approximate solution.

Here is the idea. We pick an appropriate number N , replace each value v_i with $v'_i = \lfloor v_i/N \rfloor$ and solve this modified knapsack problem. Suppose we choose N such that $\max_i v'_i \leq 2n$. Assume that each weight $w_i \leq W$.

Correction: instead of $\max_i v'_i \leq 2n$, it should have been $\max_i v'_i = 2n$. Alternately, we can say $N = (\max_i v_i)/(2n)$.

(a) [6 marks] Show that the optimal solution to the modified knapsack problem gives us an $1/2$ -approximate solution for the original problem.

Let S' be a subset of objects which forms an optimal solution for the modified knapsack problem. Let S^* be a subset of objects that forms an optimal solution for the original knapsack problem. We need to show that

$$\sum_{i \in S'} v_i \geq 1/2 \sum_{i \in S^*} v_i.$$

We can say that for any subset S with weight at most W ,

$$\sum_{i \in S'} v'_i \geq \sum_{i \in S} v'_i.$$

In particular, it's also true for an optimal solution S^* for the original problem. That is,

$$\sum_{i \in S'} v'_i \geq \sum_{i \in S^*} v'_i. \tag{1}$$

Such an inequality may not hold for given values v_i , because v'_i is some kind of an approximation for v_i . We will see that the inequality is true in an approximate sense. First observe that, for any i ,

$$v_i \geq Nv'_i = N\lfloor v_i/N \rfloor \geq N(v_i/N - 1) = v_i - N.$$

Putting this together with (1), we can write

$$\begin{aligned} \sum_{i \in S'} v_i &\geq \sum_{i \in S'} Nv'_i \\ &\geq \sum_{i \in S^*} Nv'_i \\ &\geq \sum_{i \in S^*} v_i - N|S^*| \\ &\geq \sum_{i \in S^*} v_i - Nn \\ &\geq \sum_{i \in S^*} v_i - (\max_i v_i)/2 \end{aligned}$$

The last inequality is obtained by putting $N = (\max_i v_i)/(2n)$. Now, observe that any set with single element has its weight at most W (as given in the problem). Hence, a single element set is a valid solution and the optimal solution S^* should have better value than any single element set. In particular, $\sum_{i \in S^*} v_i \geq \max_i v_i$. Plugging this in the last inequality above, we get

$$\sum_{i \in S'} v_i \geq \sum_{i \in S^*} v_i - (\sum_{i \in S^*} v_i)/2 \geq 1/2(\sum_{i \in S^*} v_i)$$

(b) [6 marks] Give an algorithm for the modified knapsack problem, whose running time is polynomial in the input size.

As mentioned, the usual DP algorithm take pseudo-polynomial time. In the modified knapsack, we have an additional assumption that $\max_i v_i = 2n$. That is, all values are integers between 0 and $2n$. We will solve the following related problem.

For a given target value V , what is minimum possible weight of a set that has value at least V ?

We will solve this problem via dynamic programming. Let A be an array such that $A(i, j)$ is supposed to be the minimum possible weight of a subset of first i objects such that the total value is at least j . If there is no such subset then the entry should be ∞ . As the base case, $A(i, 0) = 0$ for every i .

We can get a recurrence relation for $A(i, j)$ as follows. In the desired subset, either we will include i th object or not. If we include i th object then the remaining target value is $j - v_i$.

$$A(i, j) \leftarrow \min\{A(i-1, j), w_i + A(i-1, j - v_i)\}.$$

In the above equation, if $j - v_i$ is negative then replace it with zero.

We will compute the entries of A for $1 \leq i \leq n$ and $0 \leq j \leq 2n$. Then, we scan the last row i.e. $A(n, 1), A(n, 2), \dots, A(n, 2n)$. Find the largest j such that $A(n, j) \leq W$. Then j is the optimal value.

Question 8 (Curve fitting) [6 marks]. Given n points in \mathbb{R}^2 , $(p_1, q_1), (p_2, q_2), \dots, (p_n, q_n)$, we want to find a degree (at most) d polynomial $h(x) = a_0 + a_1x + a_2x^2 + \dots + a_dx^d$ so that the error

$$E(h) = \sum_{i=1}^n |h(p_i) - q_i|$$

is minimized. Write a linear program for this problem. Clearly indicate what are unknowns and what is given.

Unknowns: a_0, a_1, \dots, a_d and e_1, e_2, \dots, e_n .

-1 for not mentioning what are unknowns.

Given: p_1, p_2, \dots, p_n and q_1, q_2, \dots, q_n .

$$\begin{array}{ll} \min \sum_{i=1}^n e_i & \text{subject to} \\ e_1 & \geq a_0 + a_1p_1 + a_2p_1^2 + \dots + a_dp_1^d - q_1 \\ e_1 & \geq q_1 - (a_0 + a_1p_1 + a_2p_1^2 + \dots + a_dp_1^d) \\ e_2 & \geq a_0 + a_1p_2 + a_2p_2^2 + \dots + a_dp_2^d - q_2 \\ e_2 & \geq q_2 - (a_0 + a_1p_2 + a_2p_2^2 + \dots + a_dp_2^d) \\ & \vdots \\ e_n & \geq a_0 + a_1p_n + a_2p_n^2 + \dots + a_dp_n^d - q_n \\ e_n & \geq q_n - (a_0 + a_1p_n + a_2p_n^2 + \dots + a_dp_n^d) \end{array}$$

Question 9 (Reduction) [8 marks]. Suppose someone has written a really fast program for the independent set problem: given a graph and a number k , it finds an independent set of size k , if it exists. A set of vertices is called independent if no two vertices in the set have an edge.

We have a list of n courses offered in the next semester and there are ℓ time slots available. For each course i , its instructor has given a preferable subset L_i of slots and we are required to assign it one of the slots from L_i . We are also given a list of pairs of courses for which we need to avoid slot clash. That is, for every given pair, the two courses must get different slots. We need to find a slot allocation under these constraints, or say that it is not possible.

This is an NP-hard problem, so we cannot hope for a polynomial time algorithm. How can you use the above fast program for independent set problem to solve this?

Hint: make $|L_i|$ vertices for course i .

First we construct a graph from the given information. For any course $i \in \{1, 2, \dots, n\}$, we create vertices $\{c_{i,j} : j \in L_i\}$, that is, one vertex for each preferable slot. We make $\{c_{i,j} : j \in L_i\}$ a clique, that is, we add edges between every pair of the kind $(c_{i,j}, c_{i,j'})$.

For any given pair (i_1, i_2) of courses, where we need to avoid clash, we add edge $(c_{i_1,j}, c_{i_2,j})$ for every slot $j \in L_{i_1} \cap L_{i_2}$ (slot which is common to both courses).

Using the independent set program, we compute an independent set of size n (if one exists). From this independent set, we will compute a slot assignment as follows. For each course i , the independent set will have precisely one vertex from $\{c_{i,j} : j \in L_i\}$. This is because $\{c_{i,j} : j \in L_i\}$ is a clique, so independent set can take at most one of these vertices, and to have total n vertices, it must take one from this clique. If the independent set has vertex $c_{i,h}$ for some $h \in L_i$, then we put course i in slot h . Note that any two course i_1 and i_2 for which we wanted to avoid clash, we will not get the same slot h , because we had edge between vertices $(c_{i_1,j}, c_{i_2,j})$. That finishes the slot assignment.

Correctness: 1 mark

We have already argued that we get a valid slot assignment, if we have an independent set of size n . But what is the guarantee that the graph has an independent set of size n ? In other words, if there is no independent set of size n , then can we argue that a slot assignment for each course is not possible?

Let us argue the contrapositive. Suppose it is possible to assign one slot to each course, without any undesirable clash. Then we can construct an independent set of size n as follows: for each course i , if it is in slot h , then pick vertex $c_{i,h}$. Clearly, we have picked n vertices. Note that if two course i_1 and i_2 have the same slot h , that means this pair is not in the list of undesirable clashes. Hence, there will be no edge between $c_{i_1,h}$ and $c_{i_2,h}$. And there are no edges between any pair of vertices corresponding to different slots. Thus, the set is indeed independent.

Question 10 (Error correction). Suppose we have 36 points in the 2D plane $(1, q_1), (2, q_2), \dots, (36, q_{36})$. We consider the unique degree 35 curve $y = g(x)$ passing through these 36 points. Here $g(x)$ is a polynomial of degree at most 35. Consider 64 other points on this curve $(37, q_{37}), \dots, (100, q_{100})$. Suppose 32 out of these 100 points are modified arbitrarily in the y -coordinate (we don't know which ones), and we given these modified points $(1, q'_1), (2, q'_2), \dots, (100, q'_{100})$. We know that $q_j = q'_j$ for 68 choices of j (we don't know which ones).

Consider a degree 32 polynomial $e(x)$ which is zero at those 32 x -coordinates for which there was a modification. Clearly, $e(x)$ is unknown. Consider another polynomial $h(x) = e(x) \times g(x)$, which is also unknown.

(a) [2 marks]. Argue that for each $1 \leq j \leq 100$, we have

$$h(j) = e(j) \times q'_j.$$

By definition $h(x) = e(x) \times g(x)$. For any $1 \leq j \leq 100$, if there was no modification in q_j , then we have

$$h(j) = e(j) \times g(j) = e(j) \times q_j = e(j) \times q'_j.$$

And for any $1 \leq j \leq 100$, if there was modification in q_j , then we have $e(j) = 0$ and thus, $h(j) = 0$. Hence, $h(j) = e(j) \times q'_j$ is still valid.

(b) [4 marks]. Set up a system of 100 equations to find the (unknown) coefficients of the polynomials $h(x)$ and $e(x)$. Argue that the system is solvable.

Let

$$h(x) = h_0 + h_1x + \dots + h_{67}x^{67}.$$

$$e(x) = e_0 + e_1x + \dots + e_{32}x^{32}.$$

From part (a), we can write for any $1 \leq j \leq 100$,

$$h_0 + h_1j + \dots + h_{67}j^{67} = q'_j \times (e_0 + e_1j + \dots + e_{32}j^{32}).$$

Here the unknowns are $h_0, h_1, \dots, h_{67}, e_0, e_1, \dots, e_{32}$. That is, we have 101 unknowns. Since, there are 100 homogeneous equations, which is less than the number of unknowns, the system is solvable.

In fact, the coefficients of the polynomials $h(x)$ and $e(x)$ are indeed solutions of this system.

(c) [2 marks]. How will you find $g(x)$ after solving this system?

We will obtain the coefficients h_j s and coefficients e_j s from the solving system. We will construct polynomials $h(x)$ and $e(x)$ based on these coefficients. Finally, we will divide $h(x)$ by $e(x)$ using the standard polynomial division. That will give us $g(x)$.

(d) [4 marks]. Argue that the obtained $g(x)$ is indeed correct.

We set up a system in part(b) such that coefficients of $h(x)$ and coefficients of $e(x)$ form a solution of the system. However, when we solve the system, we might get a different solution. Depending on the rank of the linear system of equations, it may have many different solutions. So, there is no guarantee that solving the system will give us the correct $h(x)$ and $e(x)$.

Suppose we obtain some solution from the system and treating them as coefficients, we construct two polynomials, say $h^*(x)$ and $e^*(x)$. We computed $g(x)$ as $h^*(x)/e^*(x)$. Note that we do not even know if $h^*(x)$ is divisible by $e^*(x)$. We will argue that the division is possible and obtained $g(x)$ is correct, irrespective of whether $h^*(x)$ and $e^*(x)$ were same as $h(x)$ and $e(x)$ as defined earlier.

Note that since $h^*(x)$ and $e^*(x)$ are solutions of the above system, we can say that for each $1 \leq j \leq 100$, we have

$$h^*(j) = e^*(j) \times q'_j.$$

We know that for some (unknown) 68 distinct choices of j , we have $g(j) = q'_j$. Hence, for some (unknown) 68 distinct choices of j ,

$$h^*(j) = e^*(j) \times g(j).$$

Define a polynomial $Q(x) = h^*(x) - e^*(x)g(x)$. Recall that $e^*(x)$ has degree at most 32 and $h^*(x)$ has degree at most 67. Hence, $Q(x)$ has degree at most 67. Above we have concluded that $Q(j) = 0$ for 68 distinct choices of j . A degree 67 (nonzero) polynomial cannot be zero on 68 points, hence, it must be that $Q(x) = 0$ at all points. And thus, we indeed have

$$h^*(x) = e^*(x) \times g(x) \implies g(x) = h^*(x)/e^*(x).$$