Zuha Aqib 26106

Question 1

(a) *(8 points)* Construct a reduction that allows you to solve this problem efficiently using one of the algorithms we have encountered in class.

We have:

- n cards in positions $1, 2, \ldots, n$, each carrying a positive integer $l[i]$.
- A set of m *allowed pairs* (i, j), where $1 \le i < j \le n$ and $(i + j) \bmod 2 = 1$ (so each pair is one odd index and one even index).
- In one *move*, we pick an allowed pair (i, j), choose any integer $u > 1$ that divides both $l[i]$ and $l[j]$, and replace
$$l[i] \leftarrow \frac{l[i]}{u}, \quad l[j] \leftarrow \frac{l[j]}{u}.$$
- Each such division counts as *one* move, regardless of how large u is.

What is our goal? Perform the maximum number of moves before no allowed pair shares any common divisor > 1.

Key observations

Lemma 1. *Every allowed pair (i, j) always has one even and one odd index. Hence the index-graph is* bipartite*, with*

$$E = \{\textit{even positions}\}, \quad O = \{\textit{odd positions}\}.$$

Lemma 2. *The condition $(i + j) \bmod 2 = 1$ exactly means one of i, j is even, the other is odd.*

Lemma 3. *Moreover, dividing by a composite u at once never increases the total move count compared with dividing out its prime factors one at a time. Therefore we can* without loss of generality *restrict our attention to moves that divide by a* single prime *each time.*
Every composite division can be broken into single-prime divisions without reducing the move count.

Thought process
If we divide by $u = p_1^{\alpha_1} \cdots p_k^{\alpha_k}$ once, we remove α_i copies of each prime p_i. Instead, perform $\sum_i \alpha_i$ moves, each dividing by one p_i. Since each single-prime move is legal whenever the composite move was, we never lose moves—and often gain more by counting each prime separately. Concretely, if both $l[i]$ and $l[j]$ were divisible by u, they're certainly still divisible by each prime divisor of u once you peel them off one at a time.

Modeling as Bipartite Token-Matching
Fix a prime p. Write
$$l[i] = p^{e_i} \cdot r_i, \quad \gcd(p, r_i) = 1.$$

Then card i has e_i "p-tokens." A legal "divide by p" move removes one token from two cards $(i, j) \in \mathcal{P}$. Since pairs are odd–even, this is a bipartite matching between:

$$E = \{\text{even positions}\}, \quad O = \{\text{odd positions}\}.$$

Each time we divide card i by p, we consume one unit of e_i. We build a flow network $N_p = (V, E)$:

- *Vertices:* source S, sink T, one node for each $i \in E$ (even positions), and one for each $j \in O$ (odd positions).
- *Edges and capacities:*
  (a) $S \to i$ for each even i, with capacity $c(S \to i) = e_i$.
  (b) $j \to T$ for each odd j, with capacity $c(j \to T) = e_j$.
  (c) For each allowed pair (i, j) with $i \in E$, $j \in O$, add $i \to j$ with capacity $+\infty$. Infinite capacity ensures that the only limits on how many tokens pass through that edge come from the endpoints' capacities—exactly modeling "you can pair as often as both cards still have tokens."

Lemma 4. *The maximum number of single-prime-p moves equals the maximum integral flow F_p in N_p.*

Proof

(a) Flow → Moves. An integral flow of value k decomposes into k paths $S \to i \to j \to T$, each giving one legal "divide by p" on (i, j).

(b) Moves → Flow. Any valid sequence of k "divide by p" moves consumes at most e_i tokens at each i and e_j at each j, so we can send one unit of flow per move along $S \to i \to j \to T$.

Because all capacities are integers, the standard Integrality Theorem guarantees there is an optimal flow that sends whole-unit (integer) amounts along paths.
Integrality of capacities ensures an integral optimal flow.

Independence across primes

Put differently, removing a 2-token from a card has zero effect on its 3-tokens or 5-tokens, so the matchings for each prime commute.
Different primes act on disjoint token sets, so total moves is

$$\sum_{p \mid \prod_i l[i]} F_p$$

Example

Take $n = 4$ and

$$l = [12,\ 6,\ 15,\ 10], \quad \mathcal{P} = \{(1, 2), (1, 4), (3, 2), (3, 4)\}.$$

Factor exponents:

i	1	2	3	4
$l[i]$	12	6	15	10
$e_i(2)$	2	1	0	1
$e_i(3)$	1	1	1	0
$e_i(5)$	0	0	1	1

Prime 2. Even nodes $\{2,4\}$ have capacities $\{1,1\}$; odd $\{1,3\}$ have $\{2,0\}$. Allowed edges give max-flow $F_2 = 2$.

Prime 3. Even $\{2,4\}$: $\{1,0\}$; odd $\{1,3\}$: $\{1,1\}$. Max-flow $F_3 = 2$.

Prime 5. Even $\{2,4\}$: $\{0,1\}$; odd $\{1,3\}$: $\{0,1\}$. Max-flow $F_5 = 1$.
Total moves $= 2 + 2 + 1 = 5$. One valid sequence:

$$
\begin{aligned}
(1,2) \div 2 &: \ [12,6,15,10] \to [6,3,15,10],\\
(1,2) \div 3 &: \ [6,3,15,10] \to [2,1,15,10],\\
(3,4) \div 5 &: \ [2,1,15,10] \to [2,1,3,2],\\
(3,2) \div 3 &: \ [2,1,3,2] \to [2,1,1,2],\\
(1,4) \div 2 &: \ [2,1,1,2] \to [1,1,1,1].
\end{aligned}
$$

By breaking composite moves into primes, modeling each prime as a bipartite token-matching, and summing independent results, we obtain a clear, efficient, and provably optimal algorithm to maximize the number of division moves.

(b) *(7 points)* Finally, claim your title by writing code for your solution and submit it at:

`https://www.hackerrank.com/ps4-iba`

Your solution must pass **all test cases** for full credit. **Please ensure that your code submission includes your gradescope username!**

Input:

- The first line contains two integers n and m ($2 \leq n \leq 100, 1 \leq m \leq 250$) - the size of the list and the number of valid index pairs.
- The second line contains n integers $l[1], l[2], \ldots, l[n]$ ($1 \leq l[i] \leq 10^9$).
- The next m lines each contain two integers i_k and j_k ($1 \leq i_k < j_k \leq n$), such that $(i_k + j_k) \,\&\, 1 = 1$. All pairs are distinct.

Output: Output a single integer - the maximum number of operations that can be performed.

Constraints:

- $2 \leq n \leq 100$
- $1 \leq m \leq 250$
- $1 \leq l[i] \leq 10^9$
- $1 \leq i_k < j_k \leq n$
- $(i_k + j_k) \,\&\, 1 = 1$
- All pairs are distinct

Hint from TA lead Maaz on the question: What does $(i_1 + i_2) \,\&\, 1 = 1$ imply ?

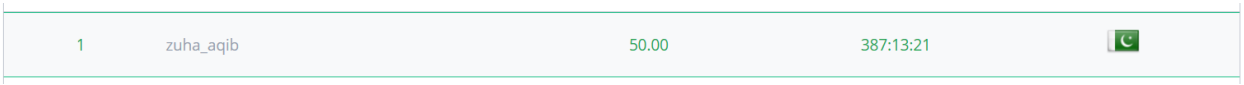

Figure 1: My submission on Hackerrank

We wish to maximize the number of times we can pick an *allowed* odd–even pair of positions (i, j) and divide both $l[i]$ and $l[j]$ by a common factor > 1. Our key steps are:

(a) **Prime-factor each input number.** Store, for each position i, a map of {prime $\mapsto$ exponent}.

(b) **Collect all primes.** The moves for distinct primes do not interfere, so we process each prime separately.

(c) **For each prime p, build a flow network:**
- Source S connects to each *even* position with capacity equal to the exponent of p at that position.
- Each *odd* position connects to sink T with capacity equal to its exponent of p.
- Each allowed even$\to$odd pair (i, j) becomes an edge of infinite capacity.

(d) **Run Dinic's max-flow** on that network; the result is the maximum number of "divide by p" moves.
(e) **Sum** the max-flows over all primes to obtain the overall maximum number of operations.

Each unit of flow from $S \to i \to j \to T$ corresponds exactly to one legal "divide by p" move on pair (i, j). By handling each prime independently, we guarantee an optimal total.

Algorithm

(a) Trial-division factorization

- We define `trial_factor(n)` to return a Python `dict` mapping each prime divisor of n to its exponent.
- We test divisors $2, 3, 5, \ldots$ up to $\lfloor\sqrt{n}\rfloor$, subtracting factors as we go, and finally handle any remainder > 1.

(b) Reading input and building adjacency

- Read n, m, then the list $L[1..n]$.
- Read the m allowed pairs (i_k, j_k). By the problem's parity hint $(i + j) \bmod 2 = 1$, one is even and one is odd.
- We store, for each even index i, a list `adj`$[i]$ of the odd indices it can pair with.

(c) Collecting prime exponents

- For each $i = 1 \ldots n$, call `trial_factor(L[i])` and store the result in `exponents`$[i]$.
- Build the set $\mathcal{P}$ of all primes seen across any $L[i]$.

(d) Dinic's max-flow class

- We maintain an adjacency list `G[u]` of [v, cap, rev] triples.
- `bfs(s,t)` builds the level graph by breadth-first search.
- `dfs(u,t,flow)` finds blocking flows in the level graph.
- `max_flow(s,t)` loops BFS+DFS until no augmenting path remains.

(e) Main prime-by-prime loop

(a) Initialize `total_moves` $= 0$.
(b) For each prime $p \in \mathcal{P}$:
  - Create a fresh `Dinic`$(n + 2)$ with vertices $\{0 = S, 1, \ldots, n, n + 1 = T\}$.
  - For each *even* i, if `exponents`$[i].get(p, 0) > 0$, add edge $(S \to i)$ with that capacity.
  - For each *odd* j, if `exponents`$[j].get(p, 0) > 0$, add edge $(j \to T)$.
  - For each allowed pair $i \to j$ in `adj`, add edge $(i \to j)$ with "infinite" capacity.
  - Compute `flow_p = max_flow(S,T)` and add to `total_moves`.
(c) Print `total_moves`.

Zuha Aqib 26106

Question 2

(a) Find a line $y = ax + c$ such that the maximum distance of any data point to the line is minimized. Your LP should have size $O(m)$.

So in this question I have a bunch of points on a 2-D plane of

$$(x^{(i)}, y^{(i)}) \in \mathbb{R}^2, \quad i = 1, \ldots, m.$$

and I want to make such a line $y = ax + c$ such that the largest possible distance of any point from the line is the smallest possible I can achieve. This is very similar to drawing a line of Best fit in our A-level physics exams. It is also similar to regression however in regression we consider all points total squared distance, here we just try to reduce the maximum one of the distances.

Optimization Function

Here we are optimizing

$$\min_{a,c} \max_{1 \le i \le m} \left| y^{(i)} - (a\, x^{(i)} + c) \right|.$$

To make it an LP problem, we are minimising the MAX distance. We can denote this max distance as t,

$$t = \max_{1 \le i \le m} \left| y^{(i)} - (a\, x^{(i)} + c) \right|$$

Thus $t \ge 0$, represents the maximum absolute residual.

Constraints

Now lets think of our constraints. Our general constraint is to reduce all distances and make them LESS THAN t:

$$\min_{a,c,t} t \quad \text{s.t.} \quad \left| y^{(i)} - (a\, x^{(i)} + c) \right| \le t \quad (i = 1, \ldots, m).$$

But in LP-problems we don't have modulus signs, so we replace this constraint with two constraints to remove the modulus signs,

$$\begin{cases} a\, x^{(i)} + c - y^{(i)} \;\le\; t, \\ y^{(i)} - \big(a\, x^{(i)} + c\big) \;\le\; t, \end{cases} \qquad i = 1, \ldots, m, \quad t \ge 0.$$

where the first line says "the vertical residual $ax^{(i)} + c - y^{(i)}$ can't exceed t."
and the second line says "the negative of that residual can't exceed t."

Final LP

Thus the full linear program (LP) is

$$\begin{aligned} \min_{a,c,t} \quad & t, \\ \text{subject to} \quad & \\ & a\, x^{(i)} + c - y^{(i)} \;\le\; t, \quad i = 1, \ldots, m, \\ & y^{(i)} - (a\, x^{(i)} + c) \;\le\; t, \quad i = 1, \ldots, m, \\ & t \;\ge\; 0. \end{aligned}$$

These two inequalities together enforce

$$|\, y^{(i)} - (a\, x^{(i)} + c)\,| \;\le\; t, \quad \forall\, i.$$

Proof that any LP solution gives a valid bound on the maximum error
We must show that solving this LP indeed finds the line $y = ax + c$ whose *maximum* vertical distance to any point is as small as possible.
Suppose (a, c, t) satisfies all LP constraints. Then for each i,

$$-\,t \;\le\; y^{(i)} - (a\, x^{(i)} + c) \;\le\; t,$$

so

$$\big|y^{(i)} - (a\, x^{(i)} + c)\big| \;\le\; t.$$

Hence t is at least as large as the largest vertical distance among all points. Since the LP minimizes t, it tries to push down that largest distance.

Proof that any best "min–max" line yields a feasible LP solution of the same cost
Let (a^*, c^*) be a line that *truly* minimizes the maximum distance:

$$T^* \;=\; \max_{1\le i\le m} \big|\, y^{(i)} - (a^* x^{(i)} + c^*)\big|.$$

Then setting $t = T^*$ makes

$$|\, y^{(i)} - (a^* x^{(i)} + c^*)| \le T^* \quad\Longrightarrow\quad (a^*, c^*, T^*) \text{ satisfies all LP constraints.}$$

Its LP objective is $t = T^*$, exactly matching the true best value.

Conclusion
The LP cannot do better than T^* because any LP solution has $t \ge T^*$.
The LP can achieve value T^* by choosing $(a^*, c^*, t = T^*)$.
Therefore the LP finds exactly the line that minimizes the maximum vertical distance.

Size of the LP

- *Variables:* a, c, t (3 total, i.e. $O(1)$).
- *Constraints:* for each of the m points there are 2 linear inequalities, plus $t \ge 0$, totaling $2m + 1 = O(m)$.
- *Encoding size:* Each inequality involves only three variables with constant-size coefficients (the $x^{(i)}, y^{(i)}$), so writing down the full constraint matrix takes

$$\underbrace{O(m)}_{\text{number of rows}} \;\times\; \underbrace{O(1)}_{\text{nonzeros per row}} \;=\; O(m)$$

  numbers. Hence the *input size* of our LP is $\Theta(m)$.
- *Running time:* By any polynomial-time LP algorithm (e.g. interior-point methods), the running time is polynomial in the *input size*. Since that size is $O(m)$, the LP can be solved in time poly(m).

Thus the LP has size proportional to m, as required.

Example

- Points: $(0,0), (1,2), (2,1)$.
- LP variables: a, c, t.
- Constraints:

$$\begin{aligned} for(0,0) &: c \le t, \quad -c \le t, \\ for(1,2) &: a + c - 2 \le t, \quad 2 - (a + c) \le t, \\ for(2,1) &: 2a + c - 1 \le t, \quad 1 - (2a + c) \le t, \\ & t \ge 0. \end{aligned}$$

- Solving (by inspection or with any LP solver) yields

$$a = 1, \quad c = 0, \quad t = 1.$$

  Indeed, the line $y = x$ has vertical distances

$$|0 - 0| = 0, \quad |2 - (1)| = 1, \quad |1 - (2)| = 1,$$

  and the worst-case distance $t = 1$ cannot be improved.

(b) Find a line $y = ax + c$ such that the sum of distances of the m data points to the line is minimized. Your LP should have size $O(m)$.

In Part (a) we minimized the *maximum* vertical distance of any data point to the line by introducing a single "worst-case" variable t and minimizing t. Here, in Part (b), we instead wish to minimize the *sum* of all vertical distances.
Consequently:

- In (a) we had one auxiliary variable t and constraints $|y^{(i)} - (ax^{(i)} + c)| \le t$ for all i, minimizing t.
- In (b) we introduce one nonnegative auxiliary v_i per data point to represent its absolute residual, enforce $|y^{(i)} - (ax^{(i)} + c)| \le v_i$, and minimize the total $\sum_i v_i$.

Formulating LP
This part is very similar to part(a) - thus its LP is also very similar.
Let the data be

$$(x^{(i)}, y^{(i)}) \in \mathbb{R}^2, \quad i = 1, \dots, m.$$

We seek parameters $a, c \in \mathbb{R}$ and *slack* variables $v_1, \dots, v_m \ge 0$ so that

$$v_i = \left| y^{(i)} - (a\,x^{(i)} + c) \right| \quad \Longrightarrow \quad \begin{cases} a\,x^{(i)} + c - y^{(i)} \le v_i, \\ y^{(i)} - (a\,x^{(i)} + c) \le v_i, \end{cases} \qquad i = 1, \dots, m.$$

The resulting linear program is:

$$\begin{aligned} \min_{a,c,\,v_1,\dots,v_m} \quad & \sum_{i=1}^{m} v_i, \\ \text{subject to} \quad & a\,x^{(i)} + c - y^{(i)} \le v_i, \quad i = 1, \dots, m, \\ & y^{(i)} - \left(a\,x^{(i)} + c\right) \le v_i, \quad i = 1, \dots, m, \\ & v_i \ge 0, \quad i = 1, \dots, m. \end{aligned}$$

We can see that it is very similar to part(a), the only difference is that there was only 1 variable there, now there are m variables.

Proof of the equivalence to the sum-of-absolute-errors objective
By definition each $v_i \ge 0$ and the two inequalities

$$-v_i \le y^{(i)} - (a\,x^{(i)} + c) \le v_i$$

force $v_i = |y^{(i)} - (a\,x^{(i)} + c)|$. Thus the LP objective

$$\min \sum_{i=1}^{m} v_i$$

exactly matches

$$\min_{a,c} \sum_{i=1}^{m} \left| y^{(i)} - (a\,x^{(i)} + c) \right|.$$

Proof of the feasibility correspondence.
Every choice (a, c) and collection of residuals

$$v_i = \left|y^{(i)} - (a\,x^{(i)} + c)\right|$$

automatically satisfies the LP constraints. Conversely, any LP-feasible $(a, c, v_1, \ldots, v_m)$ satisfies $|y^{(i)} - (a\,x^{(i)} + c)| \le v_i$, so its objective $\sum_i v_i$ is an upper bound on the true sum of residuals. Hence solving the LP yields the exact minimizer of the sum of vertical distances.

Example
Take three points:

$$(0, 0), \quad (1, 2), \quad (2, 1).$$

Introduce $v_1, v_2, v_3 \ge 0$ and constraints:

$$\begin{aligned}
a \cdot 0 + c - 0 \le v_1, &= c &\le v_1\\
0 - (a \cdot 0 + c) \le v_1, &= -c &\le v_1\\
a \cdot 1 + c - 2 \le v_2, &= a + c - 2 &\le v_2\\
2 - (a \cdot 1 + c) \le v_2, &= 2 - a - c &\le v_2\\
a \cdot 2 + c - 1 \le v_3, &= 2a + c - 1 &\le v_3\\
1 - (a \cdot 2 + c) \le v_3, &= 1 - 2a - c &\le v_3.
\end{aligned}$$

Minimizing $v_1 + v_2 + v_3$ yields the solution

$$a = 1,\ c = 0,\ v_1 = 0,\ v_2 = 1,\ v_3 = 1,$$

so the total error is $0 + 1 + 1 = 2$. Any other line gives a larger sum of vertical distances.

Size and Time Complexity Analysis

- *Variables:* $a, c,\ v_1, \ldots, v_m \implies m + 2 = O(m)$.
- *Constraints:* 2 inequalities per i + $v_i \ge 0 \implies 2m + m = 3m = O(m)$.
- *Encoding size:* each constraint involves at most three variables and constant-size data entries, so the total input description is $\Theta(m)$ numbers.
- *Solution time:* standard polynomial-time LP algorithms run in time poly(input size) = poly(m).

Hence the LP has size $O(m)$ and can be solved in time polynomial in m.

(c) Now suppose we want to fit a degree-k curve (instead of a line) to the given data. A degree-k curve is $y = c_k x^k + c_{k-1}x^{k-1} + \cdots + c_2 x^2 + c_1 x + c_0$, where the coefficients $c_i \in \mathbb{R}$. So a degree-1 curve is just a line. For a degree-k curve $y = P(x)$, the distance of $(x^{(i)}, y^{(i)})$ to the curve is $\left|P(x^{(i)}) - y^{(i)}\right|$. Find a degree-$k$ curve minimizing the sum of distances from the m data points to the curve. Your LP should have size $O(mk)$.

Building on from the previous parts,

- **Part (a)**: Fit a *line* ($k = 1$) so that the *maximum* vertical distance (the worst single error) is as small as possible. Introduced one variable t for the worst-case error and minimized t.
- **Part (b)**: Still a *line*, but now minimize the *sum* of all vertical distances. Introduced one slack variable v_i per point and minimized $\sum_i v_i$.
- **Part (c)**: Generalize to a *degree-k polynomial*

$$P(x) = c_k x^k + c_{k-1}x^{k-1} + \cdots + c_1 x + c_0,$$

and again minimize the *sum* of vertical distances. We will introduce one slack v_i per point, enforce absolute-value constraints, and minimize $\sum_i v_i$.

Problem Statement

We have m data points

$$\left(x^{(i)}, y^{(i)}\right), \quad i = 1, \ldots, m.$$

We want to choose a polynomial $P(x)$ of degree k so that if we drop each point straight up or down to the curve $y = P(x)$, the *total* of those vertical gaps is as small as possible:

$$\min_{c_0,\ldots,c_k} \sum_{i=1}^{m} \left| P(x^{(i)}) - y^{(i)}\right|.$$

Formulation of LP-problem

Introduce slack variables $v_1, \ldots, v_m \geq 0$ to represent each point's absolute error:

$$v_i = \left| P(x^{(i)}) - y^{(i)}\right| = \left| c_k(x^{(i)})^k + \cdots + c_0 - y^{(i)}\right|.$$

We enforce each absolute-value by two linear inequalities:

$$\begin{cases} c_k(x^{(i)})^k + c_{k-1}(x^{(i)})^{k-1} + \cdots + c_0 - y^{(i)} \;\leq\; v_i, \\ y^{(i)} - \left[c_k(x^{(i)})^k + \cdots + c_0\right] \;\leq\; v_i, \end{cases} \qquad i = 1, \ldots, m.$$

Then the LP is:

$$\min_{c_0,\ldots,c_k,\, v_1,\ldots,v_m} \quad \sum_{i=1}^{m} v_i,$$

subject to, for $i = 1, \ldots, m$:

$$\begin{aligned} c_k(x^{(i)})^k + \cdots + c_0 - y^{(i)} \;&\leq\; v_i, \\ y^{(i)} - \left[c_k(x^{(i)})^k + \cdots + c_0\right] \;&\leq\; v_i, \\ v_i \;&\geq\; 0. \end{aligned}$$

Proof of Feasibility and interpretation

Any choice of coefficients $c_0, \ldots, c_k$ and slacks v_i satisfying the constraints ensures

$$-v_i \;\le\; P(x^{(i)}) - y^{(i)} \;\le\; v_i \quad\Longrightarrow\quad v_i \;\ge\; \left|P(x^{(i)}) - y^{(i)}\right|.$$

Thus the LP objective $\sum_i v_i$ is an upper bound on the true sum of vertical distances.

Proof of attaining the exact error

Conversely, if we set

$$v_i^* = \left|P(x^{(i)}) - y^{(i)}\right| \quad \text{for each } i,$$

then $(c_0, \ldots, c_k, v_1^*, \ldots, v_m^*)$ satisfies all constraints, and its objective equals the true sum of errors. Therefore the LP optimizes exactly the desired quantity.

Example

Let $k = 2$ (a quadratic) and three points:

$$(0,0), \quad (1,2), \quad (2,1).$$

We introduce c_2, c_1, c_0 and $v_1, v_2, v_3 \ge 0$ with constraints:

$$\begin{aligned} c_0 &\le v_1, & -c_0 &\le v_1, \\ c_2 + c_1 + c_0 - 2 &\le v_2, & 2 - c_2 - c_1 - c_0 &\le v_2, \\ 4c_2 + 2c_1 + c_0 - 1 &\le v_3, & 1 - 4c_2 - 2c_1 - c_0 &\le v_3. \end{aligned}$$

Minimizing $v_1 + v_2 + v_3$ yields the quadratic that best fits these points in the ℓ_1 sense.

Solution by interpolation: A quadratic can pass exactly through three points, making all errors zero. Solve

$$\begin{cases} c_0 = 0, \\ c_2 + c_1 + c_0 = 2, \\ 4c_2 + 2c_1 + c_0 = 1. \end{cases}$$

From $c_0 = 0$ and $c_2 + c_1 = 2$, $2c_2 + c_1 = 0.5$. Subtracting gives

$$-c_2 = 1.5 \quad\Longrightarrow\quad c_2 = -\frac{3}{2}, \quad c_1 = 2 - c_2 = \frac{7}{2}.$$

Thus

$$P(x) = -\frac{3}{2}\,x^2 + \frac{7}{2}\,x, \quad v_1 = v_2 = v_3 = 0,$$

and the total error is $0 + 0 + 0 = 0$, which is clearly minimal.

Size and Time Complexity

- *Variables:*

$$c_0, \ldots, c_k \;(k+1) \quad + \quad v_1, \ldots, v_m \;(m) \;=\; m + (k+1) = O(m+k).$$

- *Constraints:* Two per point $\times m = 2m$, plus nonnegativity $v_i \ge 0$ adds m more, total $3m = O(m)$.

- *Nonzeros per row:* Each inequality involves the $k+1$ coefficients and one v_i, i.e. $k+2$ terms.
- *Total size:* Number of rows $\times$ nonzeros $\approx (3m)\,(k+2) = O(mk)$.
- *Solution time:* Any polynomial-time LP method runs in poly(input size) = poly(mk).

Thus, by introducing one slack variable per point and writing two simple inequalities to enforce each absolute residual, we obtain a linear program of size $O(mk)$ that exactly minimizes the sum of vertical distances to a degree-k curve. This LP can be formed in linear time and solved in time polynomial in m and k.

(d) Now imagine the data points lie in a higher dimensional space, e.g., say the data points actually are $(\mathbf{x^{(i)}}, y^{(i)})$ where now $\mathbf{x^{(i)}} \in \mathbb{R}^n$ and $y^{(i)} \in \mathbb{R}$. We now want a degree-k n-dimensional curve of the form $y = Q(x_1, x_2, \ldots, x_n)$. For example, a degree-3 curve may look like $y = 0.35x_1^3 + 3x_1^2x_3 - x_2x_{19} - 0.75x_{53} - 3x_{251} - 5$ or $y = x_1x_2^2 + 19x_7$ or $y = 7x_1x_4x_5 + 3.2x_2x_5$. The distance of data point $(\mathbf{x^{(i)}}, y^{(i)})$ to the curve is again $\left|Q(\mathbf{x^{(i)}}) - y^{(i)}\right|$. Give an LP of size at most $O(m(n+k)^k)$ size that finds an n-dimensional degree-k curve that minimizes the sum of distances of data points from the curve.

Building up from the previous parts,

- **Part (a)**: Fit a *line* ($k = 1$, one variable) to minimize the *maximum* vertical error. Introduced one variable t and minimized t.
- **Part (b)**: Fit a *line* ($k = 1$) to minimize the *sum* of vertical errors. Introduced one slack v_i per point and minimized $\sum_i v_i$.
- **Part (c)**: Fit a *univariate* polynomial of degree k in one variable, minimizing the sum of vertical errors. Introduced $k+1$ coefficients $c_0, \ldots, c_k$ plus v_i's, and minimized $\sum_i v_i$.
- **Part (d)**: Now each data point $x^{(i)}$ lives in $\mathbb{R}^n$, and we fit a *multivariate* polynomial of total degree k,
$$Q(x_1, \ldots, x_n) = \sum_{\alpha_1 + \cdots + \alpha_n \le k} c_\alpha \, x_1^{\alpha_1} \cdots x_n^{\alpha_n},$$
minimizing the *sum* of absolute vertical errors.

The new problem statement
We have m points $\left(x^{(i)}, y^{(i)}\right)$ with $x^{(i)} \in \mathbb{R}^n$ and $y^{(i)} \in \mathbb{R}$. We seek a polynomial

$$Q(x) = \sum_{|\alpha| \le k} c_\alpha \, x^\alpha \quad \text{where} \quad \alpha = (\alpha_1, \ldots, \alpha_n),\ |\alpha| = \sum_j \alpha_j \le k,$$

that makes the total of the vertical gaps

$$\sum_{i=1}^{m} \left| Q\left(x^{(i)}\right) - y^{(i)} \right|$$

as small as possible.

Intuition

i. Introduce one variable c_α for each monomial of total degree $\le k$.
ii. Introduce one nonnegative slack v_i per data point to represent $|Q(x^{(i)}) - y^{(i)}|$.
iii. Enforce $v_i \ge Q(x^{(i)}) - y^{(i)}$ and $v_i \ge y^{(i)} - Q(x^{(i)})$ by two linear inequalities.
iv. Minimize $\sum_{i=1}^m v_i$ to make the sum of errors as small as possible.

LP formulation
Let

$$Q(x^{(i)}) = \sum_{|\alpha| \le k} c_\alpha \, (x^{(i)})^\alpha.$$

Introduce $v_i \geq 0$ for $i = 1, \ldots, m$. The LP is:

$$\min_{c_\alpha, v_i} \sum_{i=1}^{m} v_i \quad \text{subject to, for each } i:$$

$$\begin{cases} Q(x^{(i)}) - y^{(i)} \leq v_i, \\ y^{(i)} - Q(x^{(i)}) \leq v_i, \\ v_i \geq 0. \end{cases}$$

This has only linear terms in the c_α and v_i.

Proof of Correctness
Any feasible (c_α, v_i) satisfies

$$-v_i \leq Q(x^{(i)}) - y^{(i)} \leq v_i,$$

so $v_i \geq |Q(x^{(i)}) - y^{(i)}|$. Hence $\sum v_i$ bounds the true total error.

Proof of Equality
Conversely, if we set

$$v_i^* = |Q(x^{(i)}) - y^{(i)}|,$$

then all constraints hold at equality, and the LP objective equals the true total error. Minimizing $\sum v_i$ therefore finds the polynomial that exactly minimizes the sum of absolute errors.

Example
$n = 2$, $k = 1$ (a bivariate linear fit) and three points:

$$x^{(1)} = (0,0),\ y^{(1)} = 1; \quad x^{(2)} = (1,0),\ y^{(2)} = 2; \quad x^{(3)} = (0,1),\ y^{(3)} = 3.$$

- Monomials of degree ≤ 1: $\{1, x_1, x_2\}$. Variables $c_{(0,0)}, c_{(1,0)}, c_{(0,1)}$.
- Slacks: $v_1, v_2, v_3 \geq 0$.
- Constraints:

$$\begin{cases} c_{00} - 1 & \leq v_1, \\ 1 - c_{00} & \leq v_1, \\ c_{00} + c_{10} \cdot 1 - 2 & \leq v_2, \\ 2 - (c_{00} + c_{10}) & \leq v_2, \\ c_{00} + c_{01} \cdot 1 - 3 & \leq v_3, \\ 3 - (c_{00} + c_{01}) & \leq v_3. \end{cases}$$

- Objective: minimize $v_1 + v_2 + v_3$.
- *Solution (by inspection):* One best choice is

$$c_{00} = 1,\ c_{10} = 1,\ c_{01} = 2, \quad v_1 = v_2 = v_3 = 0,$$

giving exact fit $Q(x_1, x_2) = 1 + x_1 + 2x_2$ through all three points.
Thus the optimisation function is $0 + 0 + 0 = 0$ which is least, and cannot be reduced further.

Size and Time Complexity

- **Coefficients:** one c_α for each monomial of degree $\leq k$, i.e. $\binom{n+k}{k}$ variables.
- **Slacks:** m variables.
- **Total variables:** $\binom{n+k}{k} + m = O(m + \binom{n+k}{k})$.
- **Constraints:** $2m$ inequalities plus m nonnegativity constraints $= 3m = O(m)$.
- **Nonzeros per constraint:** at most $\binom{n+k}{k}$ monomial terms plus one slack $= O((\binom{n+k}{k}))$.
- **Total LP size:** $O\big(m \cdot \binom{n+k}{k}\big)$.
- **Solution time:** polynomial in the input size, i.e. $\text{poly}\big(m, \binom{n+k}{k}\big)$.

By introducing one slack v_i per data point and one coefficient c_α per monomial, we obtain a linear program of size $O\big(m \binom{n+k}{k}\big)$ that exactly minimizes the sum of absolute vertical errors for a degree-k polynomial in n variables. This LP can be constructed in linear time and solved in polynomial time in m, n, and k.

Zuha Aqib 26106

Question 3

Understanding the question

So first lets understand what is a convex hull. We are given a set of points S in a $n-dimensional$ space. The convex hull of S is the smallest "rubber band shape" that stretches around all the points in S. The question says that no matter how many points there are in S, I don't need that many points to define a convex combination, I only need $n+1$ points, where n is the $n-dimensional$ space.

For example, if I am in a 2-D space and I am in a convex hull of a triangle - which is 3 points, any point inside the triangle can be written as a mixture of 3 or fewer points ($n+1=2+1=3$).

Example if I am in 3-D, I can write as at most $n+1=3+1=4$ points. For n-D, it is at most $n+1$ points.

This is a fundamental result in geometry, known formally as **Carathéodory's Theorem**.

Explaining Carathéodory's Theorem

Let $S \subseteq \mathbb{R}^n$, and let $x \in conv(S)$. $conv(S)$ means a convex combination of S.

Then, there exists a subset of at most $n+1$ points from S such that x is a convex combination of those points.

That means:

$$x = \sum_{i=1}^{k} \lambda_i x_i, \text{ where } x_i \in S,\ \lambda_i \geq 0,\ \sum \lambda_i = 1, \text{ and } k \leq n+1.$$

Any point inside the convex hull can be written as a weighted average of at most $n+1$ points. The sum of those weights should be equal to 1.

If we try to use more than $n+1$ points in $\mathbb{R}^n$, they are linearly dependent [1]. That means one of them can be written as a combination of the others. So we can remove it without changing the result. We do this repeatedly until we're left with only $n+1$ or fewer.

Step 1: Start with any convex combination

So we start off with x where x is a convex combination from set S, thus,

$$x \in \text{conv}(S)$$

by definition, there exists a finite set of points $x_1, x_2, ..., x_m \in S$, and weights $\lambda_1, \lambda_2, ..., \lambda_m \geq 0$,

[1] When at least one of the vectors from a set of vectors can be written as a linear combination of other vectors, then we can say that the vectors are linearly dependent. For example, in 2D, if you have 3 vectors and they all lie in the same plane (which they always do in 2D), then one of them must be a mix of the other two. That's linear dependence.
In $\mathbb{R}^n$, the maximum number of linearly independent vectors is n.
In $\mathbb{R}^2$: max 2 independent vectors.
In $\mathbb{R}^3$: max 3 independent vectors.
Partly cited from `https://testbook.com/maths/linearly-independent-vectors`

with $\sum_{i=1}^{m} \lambda_i = 1$ (this means that the sum of all weights λ_i is equal to 1), such that:

$$x = \sum_{i=1}^{m} \lambda_i x_i$$

Let's assume $m > n+1$ i.e., we are using more than $n+1$ points. Our goal is to show we can express x using fewer points (specifically at most $n+1$).

Step 2: Use linear dependence from $\mathbb{R}^n$
In $\mathbb{R}^n$ any set of more than $n+1$ points is linearly dependent. That means the vectors $x_1, x_2, ..., x_m$ must satisfy a non-trivial linear relation.
So, there exist scalars $\alpha_1, \alpha_2, ..., \alpha_m$, not all zero, such that:

$$\sum_{i=1}^{m} \alpha_i x_i = 0 \text{ and } \sum_{i=1}^{m} \alpha_i = 0 \ ^{2}$$

The second condition ensures that the weighted sum doesn't change the total convex combination weight (which must remain 1).

Step 3: Construct a new combination by subtracting a multiple of α
We now consider adjusting the weights λ_i slightly by defining:

$$\lambda_i' = \lambda_i - \epsilon\alpha_i$$

We choose a small enough $\epsilon > 0$ so that all $\lambda_i' \geq 0$.
Since $\sum \alpha_i = 0$, we also have

$$\sum \lambda_i' = \sum (\lambda_i - \epsilon\alpha_i) = 1$$

And:

$$\begin{aligned}
\sum \lambda_i' x_i &= \sum (\lambda_i - \epsilon\alpha_i)\ x_i \\
&= \sum \lambda_i x_i - \sum \epsilon\alpha_i x_i \\
&= \sum \lambda_i x_i - \epsilon \sum \alpha_i x_i \\
&as \sum \alpha_i x_i = 0 \\
&= \sum \lambda_i x_i - \epsilon \cdot 0 \\
&as \sum \lambda_i x_i = x \\
&= x - 0 \\
&= x
\end{aligned}$$

So we still represent the same point x, and we have a new convex combination.
Thus x is still represented, but now some $\lambda_i' = 0$ — at least one disappears. Because α is not all

[2] This has also been understood from https://www.math.cuhk.edu.hk/course_builder/1920/math4230/ch1.pdf page 8.

zeros, and we subtract $\epsilon\alpha_i$, at least one λ_i' becomes zero. This means we have eliminated one point from the combination.
We repeat this process to reduce the number of non-zero weights to at most $n+1$.

Step 4: Repeat the process

- Use linear dependence,
- Adjust weights,
- Remove a point with zero weight,

Until we're left with only $n+1$ or fewer nonzero weights.

Conclusion
Any point in the convex hull of a set $\subseteq \mathbb{R}^n$ can be written as a convex combination of at most $n+1$ points from S. The key idea is using linear dependence to eliminate redundant points while preserving the convex combination.

Zuha Aqib 26106

Question 4

Intuition

Lets start by thinking in terms of what we did in class - modelling Polynomial Multiplication. Here lets reduce this problem to Polynomial Multiplication - and model sets A and sets B as two polynomials $A(x), B(x)$ and multiply them to be $R(x)$. The coeffecients of $C(x)$ will tell us in how many ways can i sum that specific power. This seems a bit vague.
Lets write this more technically,

Understanding Polynomial Multiplication

We have two sets A and B. Each of them have n numbers. So i say lets make an $n-1$ dimension polynomial $A(x)$ where each power of x has a coffecient from set A. Repeat for set B and make polynomial $B(x)$. Now we make $C(x)$ by multiplying polynomial $A(x)$ to $B(x)$, where, each coffecient in $C(x)$ is given by,

$$C[0] = A[0].B[0]$$

where $A[0]$ is the constant in polynomial $A(x)$ and $B[0]$ is the constant in polynomial $B(x)$.

$$C[1] = A[0].B[1] + A[1].B[0]$$

where $A[1]$ is the coeffiecent of x in polynomial $A(x)$ and $B[1]$ is the coeffecient of x in polynomial $B(x)$.

$$C[2] = A[0].B[2] + A[1].B[1] + A[2].B[0]$$

where $A[2]$ is the coeffiecent of x^2 in polynomial $A(x)$ and $B[2]$ is the coeffecient of x^2 in polynomial $B(x)$.
And so on... Which gives us the formula,

$$C[i] = \sum_{j=0}^{i} A[j].B[i-j]$$

But what does this tell us? The value of $C[i]$ is the number of pairs I can find of (a, b) that sum to i where $a \in A$ and $b \in B$.

Now if I do this type of multiplication - for every index in $C(x)$ - I would need to do that many multiplications and additions. There are $2n-1$ indexes in $C(x)$ (as n in $A(x)$, same for $B(x)$, which makes $2n$ values however constant is only once so $2n-1$) so approximately

$$O(n^2) \text{ work done - the naive method}$$

Understanding FFT

To reduce it, we use FFT which computes the Discrete Fourier Transform (DFT) of a sequence efficiently. Given an input vector (polynomial coefficients) of length n, the DFT transforms it into another vector of length n — its frequency-domain representation. It will divide the input vector array into half - and recurse on each half. It uses a divide and conquer approach, by splitting in terms of even-indexed terms and odd-indexed terms. The recurrence relation is

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

(Because at the array it gets, it splits in half and recurses on both sides - it does approximate $O(n)$ work each time.) This makes the time complexity (by unrolling)

$$\text{Time complexity of } FFT : O(nlog(n))$$

Reducing the problem to Polynomial Multiplication

Now lets discuss the problem - here we have sets A and B with numbers between 0 and m. So first off we know that each set has n numbers - so lets make an n-sized array, A_a and B_a. Assign all indexes in that array with 0. Now lets traverse each element e in set A and perform the operation:

$$A_a[e] += 1$$

What is happening? We are incrementing the count of that position whenever that element occurs. Because it is not clear whether or not the $0..m$ will be repeating or unique elements in each set, or if $A \cap B = \emptyset$. So we will make the most versatile algorithm - which does not change whether or not the numbers are repeating, or unique.

So thus after traversing all the elements in set A, we now have an array A_a that has either 0's (indicates the set does not contain that index i number) and 1's (indicates that set contains that index i number).

```
# this code is not runnable and is just pseudocode

# here we define a function that will take a set and convert it to an array the way we have discussed above
def get_array(set, m):
    # start by creating an integer array of size m+1, as the max value an element can have is m. including 0 makes the size m+1
    array = size(m+1)

    # assign all values in that array with 0
    for i in range (0, m+1): # 0 is inclusive however m+1 is not
        array[i] = 0

    # now for every element in the set, traverse it and update each index accordingly. because every element in a set is unquie and non repeating, no index will have a value greater than 1.
    for element e in set:
        array[e] += 1

    return array
```

Listing 1: Tuning a set into proper format to apply FFT

Repeat for set B to build the array B_a. Now we have two arrays A_a and B_a where each index has the count of how many times that index has appeared in the set. What does this seem similar to? In polynomial multiplication, when we have two polynomials $P(x)$ and $Q(x)$, we get its points in the form of a vector-array first. And then we apply FFT. Similiarly, here we have converted our

two sets to vector-arrays just like polynomial multiplication.

Now create a third array C_a which has $2m + 1$ as each array has m elements (not including 0) and that makes $m + m = 2m$ and then one element for 0 so $2m + 1$. Run a *for loop* and assign all values 0 in C_a.

```
# this code is not runnable and is just pseudocode

def solution(setA, setB, setC, m):
    # first create two arrays, A_a and B_a from the sets A and B
    A_a = get_array(setA)
    B_a = get_array(setB)

    # now create third array C_a which will be the final solution for our
    question
    C_a = size(2m+1)
    # assign all values in this third array to be 0
    for i in range (0, 2m+2): # 0 is inclusive however 2m+2 is not
        array[i] = 0

    # perform fft, will be discussed further...

    # return whether or not we have a value a + b = c
    return false
```

Listing 2: Main Function to map our problem to FFT

So now we have successfully mapped our problem to a similar problem to FFT. We have built two indicator arrays such that

$$A_a[0..m] \text{ where } A_a[i] = 1 \text{ if } i \in A, \text{ otherwise 0. Same for } B_a$$

Applying FFT
Begin FFT, which will modify C_a. It will multiply A_a with B_a using FFT. It makes every element e in C_a equal to

$$C_a[e] = \text{ count of all possible summations of element } a \text{ from } A \text{ and } b \text{ from } B$$

What this means is that if we traverse every *possible* element in C_a and if any one of them is equal to 1, that means there exists an element $a \in A$ and $b \in B$ such that $a + b = c$ where $c \in C$.
What does *possible* mean? We are given set C, so traverse all the indexes such that $c = index \in C$.

```
# this code is not runnable and is just pseudocode

def solution(setA, setB, setC, m):
    # create sets A_a, B_a, C_a from the previous code

```

```
    C_a = perform_fft(A_a, B_a)

    # now traverse C_a for every element in setC, to find a valid
    combination for a+b=c. if it is >0, it means a combination exists thus
    return true
    for element e in setC:
        if C_a[e] > 0:
            return true

    # we reached here because we could not find a+b=c. thus return false
    return false
```

Listing 3: Updated Final Remaining Function

After applying FFT, $C_a[k]$ tells how many (a, b) pairs sum to k. So we just check if $C_a[k] > 0$ for any $c \in C$. This exactly captures the condition $a + b = c$.

Example
$A = [2, 5, 6]$
$B = [1, 3, 7]$
$C = [3, 12, 15]$
$m = 8$
So lets begin with creating indicator array A_a, of size $m + 1 = 8 + 1 = 9$, with all $0's$, $A_a = [0, 0, 0, 0, 0, 0, 0, 0, 0]$, and then traversing each element in A,
Iteration 1: element $= 2$, $A_a[2] += 1$, $A_a = [0, 0, 1, 0, 0, 0, 0, 0, 0]$
Iteration 2: element $= 5$, $A_a[5] += 1$, $A_a = [0, 0, 1, 0, 0, 1, 0, 0, 0]$
Iteration 3: element $= 6$, $A_a[6] += 1$, $A_a = [0, 0, 1, 0, 0, 1, 1, 0, 0]$
Final

$$A_a = [0, 0, 1, 0, 0, 1, 1, 0, 0]$$

now repeat to make B_a, of size 9, $B_a = [0, 0, 0, 0, 0, 0, 0, 0, 0]$,
Iteration 1: element $= 1$, $B_a[1] += 1$, $B_a = [0, 1, 0, 0, 0, 0, 0, 0, 0]$
Iteration 2: element $= 3$, $B_a[3] += 1$, $B_a = [0, 1, 0, 1, 0, 0, 0, 0, 0]$
Iteration 3: element $= 7$, $B_a[7] += 1$, $B_a = [0, 1, 0, 1, 0, 0, 0, 1, 0]$
Final

$$B_a = [0, 1, 0, 1, 0, 0, 0, 1, 0]$$

now create array C_a of size $2m + 1 = 2(8) + 1 = 16 + 1 = 17$, and initalise with all $0's$. $C_a = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$

now apply FFT, to get

$$C_a = [0, 0, 0, 1, 0, 1, 1, 1, 1, 2, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0] \ [3]$$

[3] This has been computed using FFT, for all indexes $i$in array $A_a > 0$ and for all indexes j in array $B_a > 0$, $C_a[i + j] += 1$. Example,

$$C_a[3] = 1 \ because \ 2 + 1 = 3$$

and

$$C_a[9] = 2 \ because \ 2 + 7 = 9 \ and \ 6 + 3 = 9$$

and so on.

so now traverse all elements in C to see if a addition exists,
Iteration 1: element = 3, $C_a[3] = 1$ thus > 0, exists!
Iteration 2: element = 12, $C_a[12] = 1$ thus > 0, exists!
Iteration 3: element = 15, $C_a[15] = 0$ thus < 0, does not exist

We are only trying to find if a pair exists, so for this example, we return *true*, as one pair does exist.