Assignment 01

Ishaq Hamza (22187)
Nagasai ()
Naman Mishra (22223)

Group 5

August 21, 2023

1 Part 1

Problem 1.1.

Solution. We propose the following algorithm. We assume the 3SUM oracle returns an ascending triple of indices if it finds one, and Nil otherwise. If not, we can sort the triple in $O(1)$ time.

Algorithm 1 M3SUM from 3SUM

```
external 3SUM
function M3SUM(A[1..n])
    B[1..2n] ← Transform(A)
    t ← 3SUM(B)
    if t = Nil then
        return Nil
    else
        return (t_1, t_2, t_3 − n)
where
function Transform(A[1..n])
    B[1..2n]
    for i ← 1 to n do
        B_i ← 4A_i − 1
        B_{i+n} ← 4A_i + 2
    return B
```

Suppose 3SUM on B returns (i, j, k) where $i < j < k$. Note that $B_p \equiv -1 \pmod 4$ iff $p \leq n$ and $B_p \equiv 2 \pmod 4$ iff $p > n$. Since $B_i + B_j + B_k = 0$, we have that $B_i + B_j + B_k \equiv 0 \pmod 4$.

If $n < j$, then $B_j + B_k \equiv 2+2 \equiv 0 \pmod 4$ and so $B_i \equiv 0 \pmod 4$, a contradiction.

Thus $j \leq n$ and so $B_i + B_j \equiv -2 \pmod 4$. Thus $B_k \equiv 2 \pmod 4$ and so $k > n$.

Now $B_i + B_j + B_k = 0$ gives

$$\begin{aligned} 4A_i - 1 + 4A_j - 1 + (-4)A_{k-n} + 2 = 0 \\ \implies A_i + A_j = A_{k-n} \end{aligned}$$

Since $A_p > 0$ for all p, we have that $A_i, A_j < A_{k-n}$ and so $i, j, k-n$ are all distinct.

On the other hand, if there exist distinct i, j, k such that $A_i + A_j = A_k$, then $B_i + B_j + B_{k+n} = 0$ and so 3SUM on B returns $(i, j, k+n)$. By the contrapositive, 3SUM on B fails to return a solution iff there is no solution to M3SUM on A.

Thus M3SUM $\leq$ 3SUM.

Problem 1.2.

Solution. We have the following algorithm. Let $i, j, k \in [1..n]$ be distinct. Since A

Algorithm 2 3SUM from ThreeCollinearPoints

```
1: external ThreeCollinearPoints
2: function 3SUM(A[1..n])
3:     return ThreeCollinearPoints(Transform(A))
   where
4: function Transform(A[1..n])
5:     B[1..n]
6:     for i ← 1 to n do
7:         B_i ← (A_i, A_i^3)
8:     return B
```

consist of distinct elements, $A_i - A_j, A_j - A_k, A_k - A_i$ are all non-zero. So

$$\begin{aligned} & A_i + A_j + A_k = 0 \\ \iff \quad & (A_j - A_i)(A_i + A_j + A_k) = 0 \\ \iff \quad & A_j^2 - A_i^2 + A_j A_k - A_i A_k = 0 \\ \iff \quad & A_j^2 + A_j A_k + A_k^2 = A_i^2 + A_i A_k + A_k^2 \\ \iff \quad & (A_i - A_k)(A_j^3 - A_k^3) = (A_j - A_k)(A_i^3 - A_k^3) \\ \iff \quad & \begin{pmatrix} A_i - A_k \\ A_i^3 - A_k^3 \end{pmatrix} \times \begin{pmatrix} A_j - A_k \\ A_j^3 - A_k^3 \end{pmatrix} = 0 \\ \iff \quad & (\vec{B_i} - \vec{B_k}) \times (\vec{B_j} - \vec{B_k}) = 0 \\ \iff \quad & B_i, B_j, B_k \text{ are collinear} \end{aligned}$$

where $\times$ denotes the vector cross product.

Thus every solution to 3SUM on A is a solution to ThreeCollinearPoints on B with the same set of indices and vice versa. Thus the given algorithm is correct and 3SUM $\leq$ ThreeCollinearPoints.

Problem 1.3.

Notation. $A[a_1, ..., a_2][b_1, ..., b_2]$ denotes the following subset of the array $A[m][n]$

$$\{A[i][j] \mid a_1 \leq i \leq a_2, j = 1, ..., n\} \cup \{A[i][j] \mid b_1 \leq i \leq b_2, i = 1, ..., m\}$$

Part a

Algorithm 3 $O(n \log n)$ algorithm to search for an element in a sorted row-wise and column-wise array

Require: an $n \times n$ array $A[n][n]$ of integers which is sorted row-wise and column-wise, integer x
Ensure: index (i, j) of the element x if present or NIL if absent in the array

```
1: for i ← 1 to n do
2:     t ← BINARYSEARCH(A[i][1, ..., n], x)
3:     if t ≠ −1 then
4:         return (i, t)
5: return NIL
```

Here the BINARYSEARCH function is the same as we discussed in class.
Proof of Correctness
Loop Invariant: $\mathsf{L}(i) : x \notin A[1, ..., i-1][1, ..., n]$
Base Case: $i = 1$

$$\mathsf{L}(1) : x \notin A[1, ...0][1, ..., n]$$

the above holds trivially as the array $A[1, ..., 0][1, ..., n]$ is empty.
Inductive Step: assume that $\mathsf{L}(i)$ holds before entering the loop

$$x \notin A[1, ..., i-1][1, ..., n]$$

Case 1: $x \notin A[i][1, ..., n]$

$$\begin{aligned} &\implies x \notin A[1, ..., i-1][1, ..., n] \cup A[i][1, ..., n] \\ &\implies \mathsf{L}(i+1) \end{aligned}$$

$i \mapsto i + 1$, therefore L holds after the iteration of the loop
Case 2: $x \in A[i][1, ..., n]$
binarySearch returns the index of x in $A[i][1, ..., n]$, therefore the algorithm returns the index (i, t) and the loop breaks, hence L holds vacuously

Therefore the loop invariant holds inductively.

Termination: the loop terminates when $i = n + 1$

$\mathsf{L}(n+1)$: $x \notin A[1,...,n][1,...,n]$ (which is the complete array)
$\implies$ $x \notin A$, output should be NIL.

$$\mathbb{QED}$$

Time Complexity: the loop runs for n iterations and each iteration performs a binary search on an array of size n, hence the time complexity is $O(n \log n)$

Part b

Algorithm 4 $O(m+n)$ algorithm to search for an element in a sorted row-wise and column-wise array

Require: an $m \times n$ array $A[m][n]$ of integers which is sorted row-wise and column-wise, integer x
Ensure: index (i, j) of the element x if present or NIL if absent in the array $i = m$, $j = 1$

```
1: while i > 0 and j < n + 1 do
2:     if A[i][j] < x then
3:         i − −
4:     else if A[i][j] < x then
5:         j + +
6:     else
7:         return (i, j)
```

Proof of Correctness

we define the loop invariant L as follows:

$$\mathsf{L}(i,j) : x \notin A[i+1,...,m][1,...,j-1]$$

we shall prove the invariance inductively
Base Case: $i = m, j = 1$

$$\mathsf{L}(m,1) : x \notin A[m+1,...,m][1,...,0]$$

the above statement holds trivially as in the array $A[m+1,...,m][1,..,0]$ is empty

Inductive Step: assume that $\mathsf{L}(i,j)$ and the loop condition hold before entering the loop

$$x \notin A[i+1,...,m][1,...,j-1]$$

Case 1: $A[i][j] > x$

$$\implies x < A[i][j] \leq A[i][j+1] \leq ... \leq A[i][n]$$
$$\implies x \notin A[i][j, j+1..., n]$$

this result when combined with the induction hypothesis yeilds

$$x \notin A[i+1,...,m][1,...,j-1] \cup A[i][j,j+1,...,n] \ (= A[i,...,m][1,...,j])$$

$$\implies \mathsf{L}(i+1,j)$$

$i \mapsto i+1$, therefore L holds after the iteration of the loop
Case 2: $A[i][j] < x$

$$\implies x > A[i][j] \geq A[i-1][j] \geq ... \geq A[1][j]$$
$$\implies x \notin A[1,2,...,i-1][j]$$

this result when combined with the induction hypothesis yeilds

$$x \notin A[i+1,...,m][1,...,j-1] \cup A[1,2,...,i-1][j] \ (= A[1,...,i][j,...,n])$$

$$\implies \mathsf{L}(i,j+1)$$

$j \mapsto j+1$, therefore L holds after the iteration of the loop
Case 3: if $A[i][j] = x$, then the algorithm returns the index and the loop breaks, hence L holds vacuously

Therefore the loop invariant holds inductively.

Termination: the loop terminates when either $i = 0$ or $j = n+1$
Case 1: i = 0
$\mathsf{L}(0,j)$: $x \notin A[1,...,m][1,...,j-1]$ (which is the complete array)
$\implies x \notin A$, output should be NIL.
Case 2: j = n+1
$\mathsf{L}(i,n+1)$: $x \notin A[i+1,...,m][1,...,n]$ (which is the complete array)
$\implies x \notin A$, output should be NIL.

$$\mathbb{QED}$$

Time Complexity: the loop runs for at most $m+n$ iterations and each iteration performs a constant number of comaprisons and updates, hence the time complexity is $O(m+n)$

Problem 1.4.

first we define the following function $tAsum$

Algorithm 5 $\Theta(n)$ algorithm to compute the array A'

Require: an array $A[1, ..., n]$ of integers and an integer t such that $1 \leq t \leq n$
Ensure: an array $A'[1, ..., n - t + 1]$ such that $A'[i] = A[i] + ... + A[i + t - 1]$
1: $A'[1] = A[1] + ... + A[t]$
2: **for** $i = 2$ to $n - t + 1$ **do**
3: $\quad A'[i] = A'[i - 1] - A[i - 1] + A[i + t - 1]$
4: **return** A'

Proof of Correctness
Loop Invariant: $\mathsf{L}(i) : A'[i] = A[i] + ... + A[i + t - 1]$
Base Case: $i = 1$

$$\mathsf{L}(1) : A'[1] = A[1] + ... + A[t]$$

the above holds trivially as $A'[1] = A[1] + ... + A[t]$
Inductive Step: assume that $\mathsf{L}(i)$ holds before entering the loop

$$A'[i] = A[i] + ... + A[i + t - 1]$$

$$\begin{aligned} A'[i + 1] &= A'[i] - A[i] + A[i + t] \\ &= A[i] + ... + A[i + t - 1] - A[i] + A[i + t] \\ &= A[i + 1] + ... + A[i + t] \end{aligned}$$

$i \mapsto i + 1$, therefore L holds after the iteration of the loop

Therefore the loop invariant holds inductively.

Termination: the loop terminates when $i = n - t + 1$
this is guaranteed since i is incremented by 1 in each iteration and the loop condition is $i \leq n - t + 1$
Time Complexity: the loop runs for $n - t + 1$ iterations and each iteration performs a constant number of additions and subtractions, hence the time complexity is $\Theta(n)$

Part a

Algorithm 6 $\Theta(n^3)$ algorithm to find t consecutive elements in one array whose sum is the same as the sum of t consecutive elements in the other array

Require: two arrays of integers $A[1, ..., n]$ and $B[1, ..., n]$
Ensure: (i, j, t) where $A[i]+...+A[i+t-1] = B[j], ..., B[j+t-1]$ if such subarrays exist, otherwise returns the special value NIL

```
for t = 1 to n do
    for i = n to n − t + 1 do
        A' = tAsum(A)
        B' = tAsum(B)
    for i = 1 to n do
        for j = 1 to n do
            if A'[i] = B'[j] then
                return (i, j, t)
return NIL
```

Time Complexity: in the t^{th} iteration of the outermost loop, first the array is made by calling the $tAsum$ which is $\Theta(n)$.

$$\sum_{t=1}^{n} \Theta(n) = \Theta(n^2)$$

the inner two loops comapre each pair of elements $(A'[i], B'[j])$ of the loops, hence the total number of comparisons is $\binom{n-t+1}{2}$.

$$\begin{aligned}
\sum_{t=1}^{n} \binom{n-t+1}{2} &= \sum_{t=1}^{n} \frac{(n-t+1)(n-t)}{2} \\
&= \frac{1}{2} \sum_{t=1}^{n} (n^2 - 2nt + t^2 - n + t) \\
&= \frac{1}{2} (\sum_{t=1}^{n} t^2 - (2n+1) \sum_{t=1}^{n} + \sum t = 1nn(n+1)) \\
&= \frac{1}{2} (\frac{n(n+1)(2n+1)}{6} - (2n+1)\frac{n(n+1)}{2} + n^2(n+1)) \\
&= \frac{(n-1)(n)(n+1)}{6} \\
\frac{n^3}{6} < \frac{(n-1)(n)(n+1)}{6} &< \frac{n^3}{3} \; \forall \, n > 1
\end{aligned}$$

hence the time complexity is $\Theta(n^3)$

Part b

Algorithm 7 $\Theta(n^2 \log(n))$ algorithm to find t consecutive elements in one array whose sum is the same as the sum of t consecutive elements in the other array

Require: two arrays of integers $A[1, ..., n]$ and $B[1, ..., n]$
Ensure: (i, j, t) where $A[i]+...+A[i+t-1] = B[j], ..., B[j+t-1]$ if such subarrays exist, otherwise returns the special value NIL

```
for t = 1 to n do
    A' = tAsum(A)
    B' = tAsum(B)
    sA' = sort(A')
    sB' = sort(B')
    i = 1, j = 1
    while i ≤ n − t and j ≤ n − t do
        if sA'[i] > sB'[j] then
            j++
        else if sA'[i] < sB'[j] then
            i++
        else
            return (i, j, t)
return NIL
```

Here, the sort function is assumed to work in $\Theta(n \log n)$ time.
Proof of Correctness
we define the loop invariant L as follows:

$$\mathsf{L}(i, j) : A'[a] \neq B'[b] \ \forall\, a < i,\ b < j$$

we shall prove the invariance inductively
Base Case: $i = 1, j = 1$

$$\mathsf{L}(1, 1) : A'[a] \neq B'[b] \ \forall\, a < 1,\ b < 1$$

the above statement holds trivially as the array $A'[a] \neq B'[b] \ \forall\, a < 1,\ b < 1$ is empty

Maintenance: assume that $\mathsf{L}(i, j)$ and the loop condition hold before entering the loop

$$A'[a] \neq B'[b] \ \forall\, a < i,\ b < j$$

Case 1: $A'[i] > B'[j]$

$$A'[a] \neq B'[b] \ \forall\, a < i,\ b < j + 1$$
$$\implies \mathsf{L}(i, j + 1)$$

$j \mapsto j + 1$, therefore L holds after the iteration of the loop
Case 2: $A'[i] < B'[j]$

$$A'[a] \neq B'[b] \ \forall\, a < i,\ b < j+1$$
$$\implies \mathsf{L}(i+1, j)$$

$i \mapsto i+1$, therefore L holds after the iteration of the loop
Case 3: if $A'[i] = B'[j]$, then the algorithm returns the index and the loop breaks, hence L holds vacuously

Therefore the loop invariant holds inductively.

Termination: the loop terminates when either $i = n - t + 2$ or $j = n - t + 2$
since at least one of i or j is incremented in each iteration, the loop terminates in at most $2(n - t + 1)$ iterations.
Case 1: $i = n - t + 2$
$\mathsf{L}(n - t + 2, j)$: $A'[a] \neq B'[b] \ \forall\, a < n - t + 2,\ b < j$
note that in this case, i was incremented to $n - t + 2$ only when $A'[n - t + 1] < B'[j] (\leq B[j + 1] \leq ... \leq n - t + 1)$, therefore $A'[n - t + 1] \neq B'[b] \ \forall\, b \leq n$ (since array is sorted)
$\implies A'[a] \neq B'[b] \ \forall\, a \leq n - t + 1,\ b \leq n - t + 1$
Hence we must return Nil.
Case 2: $j = n - t + 2$
$\mathsf{L}(i, n - t + 2)$: $A'[a] \neq B'[b] \ \forall\, a < i,\ b < n - t + 2$
note that in this case, j was incremented to $n-t+2$ only when $A'[i] > B'[n-t+1] (\geq B[n - t] \geq ... \geq 1)$, therefore $A'[i] \neq B'[b] \ \forall\, b \geq 1$ (since array is sorted)
$\implies A'[a] \neq B'[b] \ \forall\, a \leq n - t + 1,\ b \leq n - t + 1$
Hence we must return Nil.

$\mathbb{QED}$

Time Complexity: for each t, two calls to sort are made which are $\Theta(n \log(n))$, subsequently $tAsum$ is called, which is $\Theta(n)$
t ranges from 1 to n, hence the overall time complexity of the algorithm is $\Theta(n^2 \log(n))$

2 Part 2

Problem 2.1.

Pingala and Peasant power algorithms are the two algorithms given in the the book. They are written as recursive algorithms. We can capture the idea used in these algorithm to design a iterative algorithm to achieve the same result. The idea is to use the binary representation of the exponent to compute the result. We know that any number can be expressed as a sum of powers of 2. So, we can express the exponent as a sum of powers of 2. Then, we can use the fact that $a^{2^i} = (a^{2^{i-1}})^2$ to compute the result. The pseudo code for the above algorithm is given below.

Pseudo Code for Peasant Exponentiation

Algorithm 8 Iterative code for Peasant exponentiation

```
1:  function PeasantExponentiation(a, n)
2:      if n = 0 then
3:          return 1
4:      exponent ← n
5:      iteration ← 0
6:      base ← a
7:      if exponent%2 = 1 then
8:          result ← base
9:      else
10:         result ← 1
11:     exponent = ⌊exponent/2⌋
12:     while exponent > 0 do
13:         iteration ← iteration + 1
14:         base ← base × base
15:         if exponent%2 = 1 then
16:             result ← result × base
17:         exponent = ⌊exponent/2⌋
18:     return result
```

Observations

(i) *iteration* is the number of iterations of the while loop.

(ii) *base* is the value of a^{2^i} at the beginning of the i^{th} iteration of the while loop. As at the begining *base* is a, and it keeps getting squared. So, $base = a^{2^i}$ at the beginning of the i^{th} iteration.

(iii) *exponent* is the value of n at the beginning of the i^{th} iteration of the while loop. Moreover, the expression $exponent\%2$ at the i^{th} iteration is equal to i^{th} bit from right in binary representation of n (This is because, remainders obtained by repeated division of 2 gives the binary representation of the number).

Loop Invariant

First, let's express n in binary form as,

$$n = 2^0\lambda_0 + 2^1\lambda_1 + \ldots + 2^k\lambda_k$$

where each $\lambda_i \in \{0, 1\}$. Then loop invariant is $result = a^{2^0\lambda_0 + \ldots 2^{iteration}\lambda_{iteration}}$ after $iteration^{th}$ iteration of the while loop.

Initialization

Initially, $iteration = 0$. If $exponent\%2 = 1$, implies $\lambda_0 = 1$, otherwise $\lambda_0 = 0$. We updated the value of result accordingly. In each case, $result = a^{2^0\lambda_0}$. Hence, loop invariant is true.

Maintenance

Assume the loop invariant is true for some iteration. We need to prove that loop invariant is true for the next iteration. Suppose, the loop invariant is true for i^{th} iteration i.e

$$result = a^{2^0\lambda_0 + \ldots 2^i\lambda_i}$$

In the next iteration, *iteration* is incremented by one. So, $iteration = i + 1$. Also, *base* is squared. So, $base = a^{2^{i+1}}$. Also, *exponent* is divided by 2. So, $exponent = \lfloor \frac{n}{2^{i+1}} \rfloor$. Now, we consider the two cases. If $exponent\%2 = 0$, then $\lambda_{i+1} = 0$. So,

$$\begin{aligned} result &= a^{2^0\lambda_0 + \ldots 2^i\lambda_i} \\ &= a^{2^0\lambda_0 + \ldots 2^i\lambda_i} \times 1 \\ &= a^{2^0\lambda_0 + \ldots 2^i\lambda_i} \times a^{2^{i+1}\lambda_{i+1}} \\ &= a^{2^0\lambda_0 + \ldots 2^{i+1}\lambda_{i+1}} \end{aligned}$$

So, loop invariant is still true. If $exponent\%2 = 1$, then $\lambda_{i+1} = 1$. In this case, we are multiplying *result* by *base*. So, new *result* will be

$$\begin{aligned} result &= a^{2^0\lambda_0 + \ldots 2^i\lambda_i} \times a^{2^{i+1}} \\ &= a^{2^0\lambda_0 + \ldots 2^{i+1}\lambda_{i+1}} \end{aligned}$$

So, loop invariant is still true. Hence, in all logical flows, loop invariant is true for the next iteration.

Termination

As *exponent* is a finite number, if we keep doing floor division, it will eventually reach 0 because any strictly decreasing sequence of Natural numbers must terminate at some point. More precisely, the loop will terminate just after $iteration = k$. At this point, $exponent = 0$ so the loop terminates. So, $result = a^{2^0\lambda_0 + \ldots 2^k\lambda_k}$ (due to loop invariant). So, $result = a^n$. Hence, the algorithm is correct.

Complexity analysis

As noted earlier, the maximum value of *iteration* is k. Therefore, the loop can run atmost k times. In each iteration, we are doing a constant amount of work (addition, multiplication, modulo operation, comparisons all take $O(1)$ amount of work). So, the total work done is $O(k)$. Now, we need to find the value of k. We know that $n = 2^0\lambda_0 + 2^1\lambda_1 + \ldots + 2^k\lambda_k$. So, $k = \lfloor \log_2 n \rfloor$. So, the total work done is $O(\log n)$. Therefore, the algorithm runs in $O(\log n)$.

Pseudo Code for Pingala Exponentiation

Algorithm 9 Iterative code for Pingala exponentiation

```
1: function PingalaExponentiation(a, n)
2:     N ← n
3:     if N = 0 then
4:         return 1
5:     result ← 1
6:     mask ← GreaterPower2(N)
7:     while mask > 1 do
8:         result ← result × result
9:         mask ← mask ≫ 1
10:        if N ⊙ mask ≠ 0 then
11:            result ← result × a
12:            N ← N ⊕ mask
13:    return result
14: function GreaterPower2(n)
15:    result ← 1
16:    repeat
17:        result ← result ≪ 1
18:    until result > n
19:    return result
```

Here, $\odot$ is the bitwise AND operator, $\oplus$ is the bitwise XOR operator and $\ll, \gg$ are the left and right shift operators respectively.

This pseudo code is very similar to the previous one. We are still using the binary representation of the exponent, through the variable *mask*. We verified that the algorithm is mimicing the recursive algorithm.

Termination

The loop in GreaterPower2 terminates in less than $\log_2(n) + 2$ iterations, since *result* is doubled in each iteration.

The loop in PingalaExponentiation terminates when $mask \leq 1$. This must always happen in a finite number of iterations, since $x \gg 1 < x$ for all $x > 0$. Thus, *mask* is a strictly decreasing sequence of positive integers, and must eventually fall below 2.

Correctness

The helper function GreaterPower2 returns the lowest power of 2 greater than n. This is correct by the loop invariant

$$result \text{ is a power of two}$$

and the loop condition $result \leq n$.

For correctness of PingalaExponentiation, let I be the proposition

$$a^n = result^{mask} \cdot a^N \text{ and } mask \text{ is a power of 2 and } 0 \leq N < mask.$$

This is trivially true before the first iteration, as $N = n$ and $result = 1$, and $mask$ is the lowest power of 2 greater than n.

Suppose I is true before the k-th iteration. We consider two cases, where $mask' = mask \gg 1$ as updated on line 9. Note that $mask'$ is a power of 2, since $mask$ is a power of 2 greater than 1 (by the loop condition).

- **Case 1:** $N \odot mask' = 0$. We have $result' = result^2$ (line 8) and $N' = N$. Thus,

$$\begin{aligned}(result')^{mask'} \cdot a^{N'} &= (result^2)^{mask/2} \cdot a^N \\ &= result^{mask} \cdot a^N \\ &= a^n. \qquad \text{(by induction hypothesis)}\end{aligned}$$

  Also, since $N \geq 0$, we have $N' = N \geq 0$. Finally, since $N' \odot mask' = 0$ and $N' < 2mask'$, we have that $N' + mask' < mask' \ll 1 = 2mask'$ and so $N' < mask'$.

- **Case 2:** $N \odot mask \neq 0$. We have $result' = result^2 \cdot a$ (line 8, 11) and $N' = N \oplus mask'$ (line 12).

  Also note that $N \odot mask' \neq 0$ implies $N \oplus mask' = N - mask'$. This is because $mask'$ is a power of 2, so it has only one bit set. Thus $N \odot mask' \neq 0 \implies N \odot mask' = mask'$ and so $N \oplus mask' = N - mask'$. Thus,

$$\begin{aligned}(result')^{mask'} \cdot a^{N'} &= (result^2)^{mask/2} \cdot a^{mask'} \cdot a^{N \oplus mask'} \\ &= result^{mask} \cdot a^{mask' + N \oplus mask'} \\ &= result^{mask} a^{mask' + N - mask'} \\ &= result^{mask} \cdot a^N \\ &= a^n. \qquad \text{(by induction hypothesis)}\end{aligned}$$

  Also, since $N, mask \geq 0$, we have $N' = N \oplus mask' \geq 0$. Finally, $N' = N - mask' < mask - mask'$ (induction hypothesis) $= 2mask' - mask' = mask'$.

Thus, I is true after the k-th and before the $(k+1)$-th iteration. By induction, I is true after the last iteration.

Outside the loop (line 13), we have $mask \leq 1$ by the loop condition, and that $mask$ is a power of 2 by the loop invariant. Thus $mask = 1$.

Moreover, we have $0 \leq N < mask = 1$ by the loop invariant, so $N = 0$. Thus by the loop invariant, $a^n = result^{mask} \cdot a^N = result^1 \cdot a^0 = result$. Thus the function returns the correct value at line 13.

Complexity Analysis

In GreaterPower2, the loop runs at most $\lfloor \log_2 n \rfloor + 1$ times. Each bitwise shift costs $O(1)$ time (at least relative to multiplication), so GreaterPower2 runs in $O(\log n)$ time.

In PingalaExponentiation, in each while loop, we are doing a constant amount of work, since all bitwise operations take constant time. The while loop can run at maximum $\lfloor \log_2 n \rfloor$ times as running further would imply that $mask$ variable is halved each time. So, the total work done is $O(\log n)$. Therefore, the algorithm runs in $O(\log n)$.

Problem 2.2.

We are given with three sorted arrays $A[1 \dots p], B[1 \dots q], C[1 \dots r]$. We are given that there is atleast one common element in all the three arrays. We need to find a common element in all the three arrays. We can solve this problem in $O(n)$ time. This solution is inspired from the "TWO SUM" problem, where we use two pointers to tranverse the array. Instead of two pointers, pointing to same array, here we use three pointers, each one pointing to different array. We start with the first element of each array. We compare the three elements. If they are equal, we return the element. If they are not equal, we increment the pointer pointing to the smallest element. We repeat this process until we find the common element. The pseudo code for the above algorithm is given below.

Algorithm 10 Common Element in Three Sorted Arrays

```
1: function CommonElement(A[], B[], C[], p, q, r)
2:     first ← 0
3:     second ← 0
4:     third ← 0
5:     while first < p and second < q and third < r do
6:         maximal ← max(A[first], B[second], C[third])
7:         if A[first] = B[second] = C[third] then
8:             return (first, second, third)
9:         if A[first] < maximal then
10:            first ← first + 1
11:        if B[second] < maximal then
12:            second ← second + 1
13:        if C[third] < maximal then
14:            third ← third + 1
15:    return "No common element found"
```

Loop Invariant

Let x be the first common element present in all three elements (such an element x is guaranteed by the question itself). Let us also denote $I_D(y)$ as the index of first occurenece of y in the array D. Notice that $I_A(x) < p$ *and* $I_B(x) < q$ *and* $I_C(x) < r$.

Loop Invariant is

$$first \leq I_A(x) \quad second \leq I_B(x) \quad third \leq I_C(x)$$

Initialization

Initially, $first = second = third = 0$. So, $I_A(x), I_B(x), I_C(x) \geq 0 = first = second = third$. Hence, loop Invariant is true

Maintenance

Let us assume that the loop invariant is true for some iteration. We need to prove that the loop invariant is true for the next iteration. if $A[first], B[second], C[third]$ are all equal to each other. Then loop terminates. So, we won't deal with that here. Suppose, that they all are not equal. Then there must be a maximal element among them. WLOG, let $C[third]$ be the maximal element. If $A[first] < C[third]$, we are incrementing the left pointer by one. As $A[first] < C[third]$ and $C[third] \leq x$ (assumption that loop Invariant was true in the beginning of loop), we have $A[first] < x$. This implies, $first < I_A(x)$. So, $first + 1 \leq I_A(x)$. So, incrementing first by one, doesn't effect the loop invariant. Hence, loop Invariant is true for the next iteration. Similarly, we can prove for the case if $B[second] < C[third]$ that loop Invariant is true for the next iteration.

Termination

First, we will show that loop must terminate at some point. Then, we will show that at the point of termination, our desired outcome is reached. In each iteration, atleast one of the pointers is incremented. If the function never returns, the pointers keep on going right till they exceed the size of the array. So, the while condition will be violated (if it doesn't return by then). Hence, loop will terminate.

Now, I will show that loop will only terminate, when the required condition is satisfied i.e

$$A[first] = B[second] = C[third] = x$$

For the sake of contradiction, assume that loop terminates without satisfying the above condition. Then, it means that loop got terminated due to violation of while loop condition. This means that atleast one of the pointers has exceeded the size of the array. WLOG, let $first$ exceed the size of the array. Then, $first \geq p$. This implies, $I_A(x) \geq first \geq p$ (as loop invariant must maintained through out). This is contradiction to the fact that $I_A(x) < p$. Hence, loop will terminate only when the required condition is satisfied.

Complexity Analysis

Notice that in each iteration, we are incrementing atleast one of the pointers and each of them can't exceed their respective array lengths. So, the loop will run atmost $p + q + r$ times. In each iteration, we are doing a constant amount of work

(comparisons, addition, incrementing pointers all take $O(1)$ amount of work). So, the total work done is $O(p+q+r)$. Therefore, the algorithm runs in $O(p+q+r)$.

Problem 2.3.

Solution. We assume that the arrays are sorted in ascending order.

(i) **min $W1$**.

Algorithm 11 Minimum of W_1

```
1:  external SORT(LIST, KEY)
2:  function MINW1(A[1..n], B[1..n], C[1..n])
3:      (A, B, C) ← SORT([A, B, C], key = X ↦ X_n)
4:      p = q = r = n
5:      (a, b, c) ← (A_p, B_q, C_r)
6:      w ← ∞
7:      while r > 1 do
8:          w ← min(w, max(|a − b|, |b − c|, |c − a|))
9:          r ← r − 1
10:         ((A, p), (B, q), (C, r)) ←
                SORT([(A, p), (B, q), (C, r)], key = (X, i) ↦ X_i)
11:         (a, b, c) ← (A_p, B_q, C_r)
12:     w ← min(w, max(|a − b|, |b − c|, |c − a|))
13:     return w
```

where SORT sorts a list of three elements in ascending order according to a key. An implementation for three elements is trivial.

Proof of termination. The loop terminates when $r = 1$. In each loop, r is decremented by 1, and then p, q, r are permuted so that A_p, B_q, C_r are the largest elements. Thus $p+q+r$ decreases by 1 in each loop. Since $p+q+r$ is initially $3n$, the loop terminates after an at most $3n-2$ iterations. □

Proof of runtime. Assignments, min, max and SORT are $O(1)$ (SORT takes only 3 elements as input each time). The loop terminates in at most $3n-2$ iterations. Thus the runtime of MINW1 is $O(n)$. □

Proof of correctness. Let $I = P_1 \wedge P_2 \wedge r \geq 1$ where P_1 is the proposition that

$$a = A_p \wedge b = B_q \wedge c = C_r \wedge a \leq b \leq c$$

and P_2 is the proposition that

$$\forall\, i, j, k \in [1..n](i > p \vee j > q \vee k > r \implies w \leq \max(|A_i - B_j|, |B_j - C_k|, |C_k - A_i|)).$$

Before the loop, $p = q = r = n$ and $a = A_n, b = B_n, c = C_n$, where A, B, C are already taken such that $A_n \leq B_n \leq C_n$ (in line 3). Thus P_1 is true. Moreover, P_2 is vacuously true.

Suppose I is true before the k-th iteration of the loop. We have on line 9 that $r' = r - 1$, $c' = C[r-1]$ and $w' = \min(w, \max(|a-b|, |b-c|, |c-a|))$.

Note that since $a \leq b \leq c$, we have that

$$\begin{aligned}\max(|a-b|, |b-c|, |c-a|) &= \max(b-a, c-b, c-a) \\ &= c - a\end{aligned}$$

and so $w' = \min(w, c-a)$.

Let $i, j, k \in [1..n]$ such that $i > p \vee j > q \vee k > r'$. If $i > p$ or $j > q$, then $w' \leq w \leq \max(|A_i - B_j|, |B_j - C_k|, |C_k - A_i|)$ by the induction hypothesis.

Otherwise, $i \leq p$, $j \leq q$ and $k > r' = r - 1$. If $k > r$, then again $w' \leq w \leq \max(|A_i - B_j|, |B_j - C_k|, |C_k - A_i|)$ by the induction hypothesis. Otherwise, $k = r$. Since $C_r \geq A_p \geq A_i$ and $C_r \geq B_q \geq B_j$,

$$\begin{aligned}\max(|A_i - B_j|, |B_j - C_k|, |C_k - A_i|) &\geq \max(C_r - B_j, C_r - A_i) \\ &\geq \max(C_r - B_q, C_r - A_i) \\ &\geq \max(C_r - B_q, C_r - A_p) \\ &= \max(c-b, c-a) \\ &= c - a \\ &\geq w'\end{aligned}$$

Thus P_2 is true.

Line 9 leaves A, B, C and p, q, r, all the free variables of P_2 unchanged. Thus P_2 is still true after line 9.

Line 10 permutes $(A, p), (B, q), (C, r)$ such that $A_p \leq B_q \leq C_r$. Since it is only a permutation of terms that P_2 is symmetric in, P_2 is still true after the permutation.

Line 11 leaves A, B, C and p, q, r unchanged. Thus P_2 is still true after the k-th iteration.

Moreover, line 11 ensures that $a = A_p$, $b = B_q$, $c = C_r$, with $a \leq b \leq c$ (due to the sorting on line 10), and so P_1 is true after the k-th iteration.

Lastly, since $r' = r - 1$, the loop condition $r > 1$ ensures that $r' \geq 1$.

Thus I is true after the k-th iteration and before the $(k+1)$-th iteration. Thus by induction, I is true after the loop terminates.

Now consider line 12. $w' = \min(w, \max(|a-b|, |b-c|, |c-a|))$. The loop condition $r > 1$ is false, so $r \leq 1$. By the loop invariant, $r \geq 1$ and so $r = 1$.

Let $i, j, k \in [1..n]$. We have by the loop invariant that if $i > p$ or $j > q$ or $k > 1$ then $w' \leq w \leq \max(|A_i - B_j|, |B_j - C_k|, |C_k - A_i|)$.

Otherwise, $i \leq p$ and $j \leq q$ and $k = 1$. Since $C_1 \geq A_p \geq A_i$ and $C_1 \geq B_q \geq B_j$,

$$\begin{aligned}\max(|A_i - B_j|, |B_j - C_k|, |C_k - A_i|) &\geq \max(C_1 - B_j, C_1 - A_i) \\ &\geq \max(C_1 - B_q, C_1 - A_i) \\ &\geq \max(C_1 - B_q, C_1 - A_p) \\ &= \max(c - b, c - a) \\ &= c - a \\ &\geq w'\end{aligned}$$

Thus, we have for any $i, j, k \in [1..n]$ that

$$w' \leq \max(|A_i - B_j|, |B_j - C_k|, |C_k - A_i|).$$

Since w is always assigned a value of the form $\max(|A_i - B_j|, |B_j - C_k|, |C_k - A_i|)$, we have that

$$w' = \min\{\max(|A_i - B_j|, |B_j - C_k|, |C_k - A_i|) \mid i, j, k \in [1..n]\}.$$

Thus the value returned by the function on line 13 is correct. □

(ii) **max $W1$**.

Algorithm 12 Maximum of W_1

```
1: function MaxW1(A[1..n], B[1..n], C[1..n])
2:     return max{MaxPair(A, B), MaxPair(B, C), MaxPair(C, A)}
3: function MaxPair(X[1..n], Y[1..n])
4:     return max{|X_n − Y_1|, |Y_n − X_1|}
```

Proof of correctness. We first prove the correctness of MaxPair, that is,

$$\forall\, i, j \in [1..n], |X_i - Y_j| \leq \textsc{MaxPair}(X, Y).$$

Let $i, j \in [1..n]$ be arbitrary. We have two cases.

- **Case 1:** $X_i < Y_j$.
  Then $Y_n - X_1 \geq Y_j - X_1 \geq Y_j - X_i > 0$.
  So $|X_i - Y_j| \leq |Y_n - X_1| \leq \textsc{MaxPair}(X, Y)$.
- **Case 2:** $X_i \geq Y_j$.
  Then $X_n - Y_1 \geq X_i - Y_1 \geq X_i - Y_j \geq 0$.
  So $|X_i - Y_j| \leq |X_n - Y_1| \leq \textsc{MaxPair}(X, Y)$.

Thus, $|X_i - Y_j| \leq \textsc{MaxPair}(X, Y)$ for all $i, j \in [1..n]$.

We now prove that the function MaxW1 returns the correct value. Let $i, j, k \in$

$[1..n]$. Then

$$\begin{aligned}
\mathrm{W1}(i,j,k) &= \max\{|A_i - B_j|, |B_j - C_k|, |C_k - A_i|\} \\
&\geq \max\{\max\{|a-b| \mid a \in A, b \in B\}, |B_j - C_k|, |C_k - A_i|\} \\
&\geq \max\{\max\{|a-b| \mid a \in A, b \in B\}, |B_j - C_k|, |C_k - A_i|\} \\
&= \max\{\textsc{MaxPair}(A,B), |B_j - C_k|, |C_k - A_i|\} \\
&\geq \max\{\textsc{MaxPair}(A,B), \textsc{MaxPair}(B,C), |C_k - A_i|\} \\
&\geq \max\{\textsc{MaxPair}(A,B), \textsc{MaxPair}(B,C), \textsc{MaxPair}(C,A)\} \\
&= M.
\end{aligned}$$

Since $\textsc{MaxPair}(X,Y) = |x-y|$ for some $x \in X, y \in Y$, we have that M is of the form $|a-b|$ or $|b-c|$ or $|c-a|$ for some $a \in A, b \in B, c \in C$.

Thus, $M = \max \mathrm{W1}(i,j,k)$ and so the function returns the correct value. □

Proof of termination/runtime. The runtime of MaxPair is $O(1)$. Thus the runtime of MaxW1 is $O(1)$. □

(iii) **min *W2***.

Algorithm 13 Minimum of W_2

```
1: function MinW2(A[1..n], B[1..n], C[1..n])
2:     M ← min{MinPair(A, B), MinPair(B, C), MinPair(C, A)}
3:     return M
4: function MinPair(X[1..n], Y[1..n])
5:     (i, j) ← (1, 1)
6:     m ← |X_i − Y_j|
7:     while i ≤ n and j ≤ n do
8:         (i, j) ← { (i + 1, j)   if X_i ≤ Y_j
                    { (i, j + 1)   if Y_j < X_i
9:         m ← min{m, |X_i − Y_j|}
10:    return m
```

Proof of termination. MinW2 consists of three calls to MinPair. Thus MinW2 terminates if MinPair always terminates.

MinPair terminates because one of i and j is incremented in each iteration of the loop, and i and j are bounded by n. Thus the loop terminates after at most $2n-1$ iterations. □

Proof of correctness. We first prove the correctness of MinPair, that is,

$$\forall\, i, j \in [1..n], |X_i - Y_j| \geq \textsc{MinPair}(X,Y)$$

Let I be the proposition that

$$\begin{aligned}
& m = \min\{|x-y| \mid x \in X[1..i], y \in Y[1..j]\} \\
\wedge\ & X_i \geq \max Y[1..j-1] \\
\wedge\ & Y_j \geq \max X[1..i-1].
\end{aligned}$$

Clearly I is true before the first iteration of the loop, since $i = j = 1$ and $m = |X_1 - Y_1|$, and $X[1..i-1] = Y[1..j-1] = [\,]$.

Suppose I is true before the k-th iteration of the loop. Let i_0, j_0, m_0 be the values of i, j, m before the k-th iteration. Let i_1, j_1, m_1 be the values of i, j, m after the k-th iteration. We have two cases.

- **Case 1:** $X_{i_0} \leq Y_{j_0}$.
  Then $i_1 = i_0 + 1$, $j_1 = j_0$ and $m_1 = \min\{m_0, |X_{i_0+1} Y_{j_0}|\}$. Let $i \in [1..i_0 + 1]$ and $j \in [1..j_0]$.

  If $i \in [1..i_0]$ then $m_1 \leq m_0 \leq |X_i - Y_j|$ by the induction hypothesis.

  If $i = i_0 + 1$, then we have two subcases.

  - **Case 1.1:** $j = j_0$.
    Then $m_1 = \min\{m_0, |X_i - Y_j|\} \leq |X_i - Y_j|$.
  - **Case 1.2:** $j < j_0$.
    Then $m_1 \leq m_0 \leq |X_{i_0} - Y_j| = X_{i_0} - Y_j \leq X_i - Y_j$, since $X_{i_0} \geq \max Y[1..j-1]$ and X is increasing.

  Since all possible values of m_1 are of the form $x - y$ for some $x \in A[i..i_1], y \in B[1..j_1]$,
  $$m_1 = \min\{|x - y| \mid x \in A[1..i_1], y \in B[1..j_1]\}.$$
  Moreover,
  $$X_{i_1} \geq X_{i_0} \geq \max Y[1..j_0 - 1] = \max Y[1..j_1 - 1]$$
  and
  $$Y_{j_1} = Y_{j_0} \geq X_{i_0} = \max X[1..i_0] \geq \max X[1..i_1 - 1].$$

- **Case 2:** $Y_{j_0} < X_{i_0}$.
  This is identical to case 1 with the roles of X and Y as well as i and j swapped.

  $i_1 = i_0$, $j_1 = j_0 + 1$ and $m_1 = \min\{m_0, |X_{i_0} - Y[j_0 + 1]|\}$. Let $i \in [1..i_0]$ and $j \in [1..j_0 + 1]$.

  If $j \in [1..j_0]$ then $m_1 \leq m_0 \leq |X_i - Y_j|$ by the induction hypothesis.

  If $j = j_0 + 1$, then we have two subcases.

  - **Case 2.1:** $i = i_0$.
    Then $m_1 = \min\{m_0, |X_i - Y_j|\} \leq |X_i - Y_j|$.
  - **Case 2.2:** $i < i_0$.
    Then $m_1 \leq m_0 \leq |X_i - Y_{j_0}| = Y_{j_0} - X_i \leq Y_j - X_i$, since $Y_{j_0} \geq \max X[1..i-1]$ and Y is increasing.

  Since all possible values of m_1 are of the form $x - y$ for some $x \in A[i..i_1], y \in B[1..j_1]$,
  $$m_1 = \min\{|x - y| \mid x \in A[1..i_1], y \in B[1..j_1]\}.$$

Moreover,

$$X_{i_1} = X_{i_0} \geq Y_{j_0} \geq \max Y[1..j_0] \geq \max Y[1..j_1 - 1]$$

and

$$Y_{j_1} \geq Y_{j_0} \geq \max X[1..i_0 - 1] = \max X[1..i_1 - 1].$$

Thus I is true after the k-th and before the $(k+1)$-th iteration of the loop. Since I is true before the first iteration of the loop, by induction I is true after every iteration of the loop.

Outside the loop (line 10), $i > n$ or $j > n$ and by the loop invariant, $m = \min\{|x - y| \mid x \in A[1..i], y \in Y[1..j]\}$.

Suppose $i > n$. Then $y \geq Y_j \geq \max A[1..i-1] = \max A[1..n]$ for all $y \in B[j+1..n]$, and so $m \leq |x - Y_j| = Y_j - x \leq y - x = |x - y|$ for all $y \in B[j + 1..n]$ and $x \in A[1..n]$.

For $y \in B[1..j]$, $m \leq \{|x - y| \mid x \in A[1..i] = A[1..n], y \in B[1..j]\} \leq |x - y|$ for all $y \in B[1..j]$ and $x \in A[1..n]$.

Thus $m \leq |x - y|$ for all $x \in A[1..n]$ and $y \in B[1..n]$. Thus

$$m = \min\{|x - y| \mid x \in A[1..n], y \in B[1..n]\}$$

and the function MinPair returns the correct value.

The case for $j > n$ is identical.

We now prove that the function MinW2 returns the correct value. Let $i, j, k \in [1..n]$. Then

$$\begin{aligned}
\mathrm{W2}(i,j,k) &= \min\{|A_i - B_j|, |B_j - C_k|, |C_k - A_i|\} \\
&\geq \min\{\min\{|a - b| \mid a \in A, b \in B\}, |B_j - C_k|, |C_k - A_i|\} \\
&\geq \min\{\min\{|a - b| \mid a \in A, b \in B\}, |B_j - C_k|, |C_k - A_i|\} \\
&= \min\{\textsc{MinPair}(A, B), |B_j - C_k|, |C_k - A_i|\} \\
&\geq \min\{\textsc{MinPair}(A, B), \textsc{MinPair}(B, C), |C_k - A_i|\} \\
&\geq \min\{\textsc{MinPair}(A, B), \textsc{MinPair}(B, C), \textsc{MinPair}(C, A)\} \\
&= M.
\end{aligned}$$

Since $\textsc{MinPair}(X, Y) = |x - y|$ for some $x \in X, y \in Y$, we have that M is of the form $|a - b|$ or $|b - c|$ or $|c - a|$ for some $a \in A, b \in B, c \in C$.

Thus, $M = \min \mathrm{W2}(i, j, k)$ and so the function returns the correct value. □

Proof. The runtime of MinPair is $O(n)$, since the loop terminates in at most $2n-1$ iterations and each iteration takes $O(1)$ time.

Since MinW2 calls MinPair 3 times, the runtime of MinW2 is $O(3n) = O(n)$. □

(iv) **max $W2$**.

Algorithm 14 Maximum of W_2

```
1: function maxW2(A[1..n], B[1..n], C[1..n])
2:     W ← max{F(A, B, C), F(A, C, B), F(B, A, C),
           F(B, C, A), F(C, A, B), F(C, B, A)}
3:     return W
4: function F(X[1..n], Y[1..n], Z[1..n])
5:     j ← 1
6:     W ← Y_j
7:     for j ← 2 to n do
8:         W ← max(W, min(|X_1 − Y_j|, |Y_j − Z_n|, |Z_n − X_1|))
9:     return W
```

Clearly F($X[1..n]$, $Y[1..n]$, $Z[1..n]$) returns the maximum of the set

$$\{\min(|X_1 - y|, |y - Z_n|, |Z_n - X_1|) \mid y \in Y[1..n]\}.$$

This can be easily seen by considering the loop invariant I:

$$W = \max\{\min(|X_1 - y|, |Y_j - Z_n|, |Z_n - X_1|) \mid Y \in Y[1..j]\}.$$

and termination can be easily shown by considering the strictly decreasing but non-negative value of $n - j$.

We thus shift our attention to maxW2($A[1..n], B[1..n], C[1..n]$).

Proof of termination. A call to maxW2 simply calls F six times, and computes the maximum of the results. Since F terminates, so does maxW2. □

Proof of runtime. Since F runs in $O(n)$ time (the loop runs $n - 1$ times), maxW2 runs in $O(n)$ time. □

Proof of correctness. Let

$$W_2(i, j, k) = \min(|A_i - B_j|, |B_j - C_k|, |C_k - A_i|).$$

Let i, j, k be such that $A_1 \leq A_i \leq B_j \leq C_k \leq C_n$. Then

$$\begin{aligned}\min(|A_i - B_j|, |B_j - C_k|, |C_k - A_i|) &= \min(B_j - A_i, C_k - B_j, C_k - A_i)\\ \leq \min(B_j - A_1, C_n - B_j, C_n - A_1)& \\ &= \min(|A_1 - B_j|, |B_j - C_n|, |C_n - A_1|)\\ &= W_2(1, j, n)\end{aligned}$$

Thus $W_2(i, j, k) \leq W_2(1, j, n)$.

Similarly for other $i, j, k \in [1..n]$, we have indices i', j', k' such that $W_2(i, j, k) \leq W_2(i', j', k')$ with one of i', j', k' being 1 and one being n. Thus, the maximum of the set

$$\{W_2(i, j, k) \mid i, j, k \in [1..n]\}$$

is the maximum of the set

$$\{W_2(J) \mid J \text{ is a permutation of } (1, j, n) \text{ for some } j \in [1..n]\}$$

Since $\textsc{f}(X[1..n], Y[1..n], Z[1..n])$ returns the maximum of the set

$$\{\min(|X_1 - y|, |y - Z_n|, |Z_n - X_1|) \mid y \in Y[1..n]\}$$

we have that

$$\max\{\textsc{f}(\sigma) \mid \sigma \text{ is a permutation of } (A, B, C)\}$$

is the maximum of the set

$$\{W_2(J) \mid J \text{ is a permutation of } (1, j, n) \text{ for some } j \in [1..n]\}$$

and thus gives the maximum of

$$\{W_2(i, j, k) \mid i, j, k \in [1..n]\}$$

as desired. □

We can also replace the linear search in f with a binary search for the element closest to $(X_1 + Z_n)/2$ in Y, which would reduce the runtime of f to $O(\log n)$, and thus the runtime of maxW2 to $O(\log n)$.