This assignment has in total 100 base points and 10 extra points, and the cap is 100. Bonus questions are indicated using the $\star$ mark.

Submission Instructions: Please submit to Gradescope. During submission, you need to **mark/map the solution to each question**; otherwise, we may apply a penalty.

Another notice: you only get full credits for the algorithm design questions if your algorithm matches the desirable complexity. If the desirable complexity is not stated, you need to design an algorithm to be as fast as possible.

Please specify the following information before submission:

- Your Name: Yixia Yu
- Your NetID: yy5091

Problem 1: Packing heavy unit intervals [15 pts]

Suppose we are given a set $P = \{x_1, \ldots, x_n\}$ of n points on the x-axis and a positive integer k. We say an interval on the x-axis is *k-heavy* if it contains at least k points in P. We consider *left-open right-closed unit intervals*, i.e., intervals of the form $I = (a, b]$ where $b - a = 1$ (for convenience, below we simply call them *unit intervals*). Our goal is to find a maximum number of k-heavy unit intervals on the x-axis that are *disjoint* from each other. For example, if $P = \{-\frac{2}{3}, 0, \frac{2}{3}\}$ and $k = 1$, then one optimal solution is $I_1 = (-1.6, -0.6]$, $I_2 = (-0.6, 0.4]$, $I_3 = (0.5, 1.5]$, which consists of three disjoint k-heavy unit intervals (note that the optimal solution is not necessarily unique). Design a greedy algorithm HeavyInt(n, P, k) to solve this problem, which should output the set of intervals you find. First describe your greedy strategy and prove its correctness. Then implement your algorithm by giving the pseudocode. Your algorithm is supposed to run in $O(n)$ time, assuming the points in P have already been sorted in the left-right order.

Solution. Only at points $a = x_i$ or $a = x_i - 1$ (for $x_i \in P$) does the count in $(a, a + 1]$ change. Thus, define

$$Q = \{x_i - 1 : x_i \in P\} \cup \{x_i : x_i \in P\}.$$

Initialize $L = \min(P) - 1$ (the minimum allowed left endpoint) and scan Q for the smallest candidate $a \geq L$ such that the count of points in $(a, a + 1]$ is at least k. Once such an a is found, choose that interval, update L to $a + 1$, and repeat.
Correctness: By always choosing the leftmost feasible candidate:

- Each interval is heavy (i.e. contains at least k points).
- Intervals do not overlap (since the next interval's left endpoint is at least the previous one's right endpoint).
- An exchange argument shows that shifting intervals left does not reduce their heavy property, ensuring this greedy strategy is optimal.

```
1  HEAVY-INT(P, k)
2      S <- {}
3      L <- min(P) - 1
4      Q <- merge({x - 1 for x in P}, {x for x in P})
5      p_left <- 1, p_right <- 1
6      for each a_candidate in Q where a_candidate >= L do
7          while p_left <= n and P[p_left] <= a_candidate do
8              p_left <- p_left + 1
9          while p_right <= n and P[p_right] <= a_candidate + 1 do
```

```
            p_right <- p_right + 1
        if (p_right - p_left) >= k then
            a_star <- a_candidate
            break
        end if
    end for
    if no candidate found then
        return S
    while true do
        S <- S  { (a_star, a_star+1] }
        L <- a_star + 1
        Advance candidate pointer in Q to first a_candidate >= L
        found <- false
        for each a_candidate in Q where a_candidate >= L do
            Update p_left, p_right for a_candidate
            if (p_right - p_left) >= k then
                a_star <- a_candidate
                found <- true
                break
            end if
        end for
        if not found then
            break
    end while
    return S
```

Problem 2: Sorting with arbitrary swaps $[10 + 10 + 10^\star$ pts]

Given an array $A[1 \dots n]$ of distinct numbers, we want to sort it in increasing order. However, we are only allowed to change A by swapping two elements (not necessarily adjacent) in A. Specifically, we are provided an oracle SWAP. If we call SWAP(i, j), then $A[i]$ and $A[j]$ will be swapped in A.

(1) Design a simple algorithm that sorts A by calling the SWAP oracle at most $n - 1$ times. Describe the basic idea and give the pseudocode, and analyze the number of oracle calls. Your algorithm is supposed to run in $O(n \log n)$ time (and call SWAP at most $n - 1$ times), assuming each oracle call takes $O(1)$ time.

(2) Next, we consider the problem of sorting A using *minimum* number of calls of the SWAP oracle. Prof. Greed gives the following greedy algorithm for this problem. Let $\#\mathsf{Inv}(A)$ denote the number of inversions in A. The algorithm simply repeats the following step until A is sorted: find indices $i, j \in \{1, \dots, n\}$ such that swapping $A[i]$ and $A[j]$ makes $\#\mathsf{Inv}(A)$ decrease the most, and then call SWAP(i, j). Give a counterexample for which this algorithm fails to give an optimal solution (and briefly argue why it is a counterexample). To get the full credit, your example should make the algorithm fail no matter how it breaks the ties.

(c*) Design an algorithm that sorts A using minimum number of SWAP calls. Describe the basic idea and give the pseudocode. Prove the correctness of your algorithm (i.e., it successfully sorts A and the number of SWAP calls used is minimum). Your algorithm is supposed to run in $O(n^2)$ time or faster.

Solution. **Idea:**

- Make a copy B of A and sort B (in $O(n \log n)$ time) so that B holds the correct (increasing) order.
- Construct a mapping T that maps each element to its current position in A. For each index i, if $A[i] \neq B[i]$, then the element that belongs at position i is currently at position $T[B[i]]$. Call SWAP$(i, T[B[i]])$ and update the mapping. Each swap places one or two elements in their correct positions.

Since each cycle of misplaced elements of length L is fixed with $L - 1$ swaps, the total number of swaps is at most $n - 1$.

```
1        FUNCTION MIN_SWAPS(A):
2            n <- LENGTH(A)
3            B <- COPY(A)
4            quicksort(B)
```

```
5               T <- EMPTY_MAP
6               FOR i <- 0 TO n-1 DO:
7                   T[A[i]] <- i
8               END FOR
9               swaps <- 0
10              FOR i <- 0 TO n-1 DO:
11                  IF A[i] != B[i] THEN:
12                      j <- T[B[i]]
13                      SWAP(A[i], A[j])
14                      T[A[i]] <- i
15                      T[A[j]] <- j
16                      swaps <- swaps + 1
17                  END IF
18              END FOR
19              RETURN swaps
20          END FUNCTION
```

(2) **Counterexample:** Consider the array

$$A = [6, 4, 7, 1, 5, 2, 3]$$

with the target sorted order

$$[1, 2, 3, 4, 5, 6, 7].$$

Optimal Swap Count: By performing a cycle decomposition:

- Positions 1, 2, 4, and 6 form a 4-cycle (since $6 \to 2$, $2 \to 4$, $4 \to 1$, $1 \to 6$) needing $4 - 1 = 3$ swaps.
- Positions 3 and 7 form a 2-cycle (since $7 \to 3$, $3 \to 7$) needing $2 - 1 = 1$ swap.

Thus, the optimal total is $3 + 1 = 4$ swaps.

Greedy Inversion Reduction: The inversion count of A is 14. A detailed check confirms:

- Element 6 creates 5 inversions,
- Element 4 creates 3 inversions,
- Element 7 creates 4 inversions,
- Element 5 creates 2 inversions.

A candidate swap that reduces the inversion count by the most is swapping the elements at positions 1 and 7 (swapping 6 and 3). After the swap,

$$A' = [3, 4, 7, 1, 5, 2, 6],$$

the inversion count drops from 14 to 9 (a decrease of 5 inversions), which is higher than the drop achieved by swaps within a single cycle (typically 3 or less).

Global Impact: Despite the significant inversion drop, swapping positions 1 and 7 merges the initial two cycles into one 6-cycle. This new cycle requires

$$6 - 1 = 5 \text{ swaps},$$

and including the applied swap, totals 6 swaps—worse than the optimal 4. Thus, even though the (1,7) swap is locally optimal (with the largest inversion reduction), it leads to a globally suboptimal solution. This counterexample shows that relying solely on the maximum inversion drop can force the greedy algorithm into a wrong decision.

(3) **Basic Idea:** The algorithm finds cycles in the permutation of the array. Every element not in its correct position is part of a cycle. For a cycle of length k, exactly $k-1$ swaps are needed. Thus, if the array decomposes into cycles with lengths $k_1, k_2, \ldots, k_m$, the total minimum swaps is

$$(k_1 - 1) + (k_2 - 1) + \cdots + (k_m - 1)$$

which is equivalent to $n - m$ where n is the number of elements and m is the number of cycles.

```
FUNCTION MIN_SWAPS(arr)
    n <- LENGTH(arr)
    visited <- ARRAY of n Booleans (all set to FALSE)
    pos <- EMPTY HASH MAP
    FOR i <- 0 TO n - 1 DO
        pos[arr[i]] <- i
    END FOR
    SORT(arr)
    swaps <- 0
    FOR i <- 0 TO n - 1 DO
        IF visited[i] = TRUE OR pos[arr[i]] = i THEN
            CONTINUE TO NEXT i
        END IF
        j <- i
        cycleSize <- 0
        WHILE visited[j] = FALSE DO
            visited[j] <- TRUE
            j <- pos[arr[j]]
            cycleSize <- cycleSize + 1
        END WHILE
        IF cycleSize > 0 THEN
            swaps <- swaps + (cycleSize - 1)
        END IF
    END FOR
    RETURN swaps
END FUNCTION
```

Correctness Proof:

Sorting: The algorithm detects cycles in the array's permutation. Every element not in its correct position is part of a cycle of interdependent elements. By traversing each cycle using a visited array, the algorithm places each element into its correct position. When all cycles are processed, the array is fully sorted.

Minimal Swap Count: For a cycle of length k, exactly $k-1$ swaps are necessary because each swap fixes one element, and no cycle can be sorted with fewer than $k-1$ swaps. Since the cycles are disjoint, summing $(k-1)$ over all cycles yields the minimum number of swaps required. This is exactly the count produced by the algorithm.

Problem 3: Longest doubling subsequence [15 pts]

Recall that an array (or sequence) A of real numbers is *doubling* if $A[i] \geq 2A[i-1]$ for all $i \geq 2$ (thus any array of length 1 is doubling). Given an array A, we want to find a longest subsequence of A that is doubling. For example, if $A = [7, 1, 3, 8, 2, 4, 5, 6, 9]$, then a longest doubling subsequence of A is be $[1, 2, 4, 9]$, which is of length 4. Design an algorithm $\text{LDS}(n, A)$ that returns a longest doubling subsequence A' of the array A (where n is the size of A). Describe the basic idea of your algorithm and give the pseudocode. Briefly justify its correctness and analyze its time complexity. Your algorithm should run in $O(n^2)$ time or faster.

Solution. Let A be an array of n real numbers. An array is *doubling* if

$$A[i] \geq 2A[i-1] \quad \text{for all } i \geq 2.$$

Our goal is to find a longest subsequence A' of A that is doubling.

We define a subproblem for each index i as follows: let $dp[i]$ be the length of the longest doubling subsequence ending at index i. The dynamic programming recurrence is given by

$$dp[i] = \max\left\{1, \max_{\substack{1 \leq j < i \\ A[i] \geq 2A[j]}} \{dp[j] + 1\}\right\}.$$

Here, the inner maximum considers only those indices j with $1 \leq j < i$ such that the doubling condition $A[i] \geq 2A[j]$ holds. In addition, we maintain an array *pred* to record the predecessor of each element in the subsequence so that the subsequence can be reconstructed.

Once all subproblems are solved, the overall longest doubling subsequence is obtained by selecting the maximum value among all $dp[i]$ and backtracking using the *pred* array.

Pseudocode

```
1   FUNCTION LDS(n, A):
2       for i from 1 to n do:
3           dp[i] <- 1
4           pred[i] <- null
5       best <- 1
6       index <- 1
7       for i from 1 to n do:
8           for j from 1 to i-1 do:
9               if A[i] >= 2 * A[j] and dp[j] + 1 > dp[i] then:
10                  dp[i] <- dp[j] + 1
11                  pred[i] <- j
12              end if
```

```
            end for
            if dp[i] > best then:
                best <- dp[i]
                index <- i
            end if
        end for
        S <- empty sequence
        while index is not null do:
            prepend A[index] to S
            index <- pred[index]
        end while
        return S
    END FUNCTION
```

Correctness Justification

- For each i, $dp[i]$ correctly represents the longest doubling subsequence ending at $A[i]$ considering only predecessors j with $A[i] \geq 2A[j]$.
- The recurrence
$$dp[i] = \max\left\{1, \max_{\substack{1 \leq j < i \\ A[i] \geq 2A[j]}} \{dp[j] + 1\}\right\}$$
guarantees that $A[i]$ either starts a new subsequence or extends a valid one.
- The *pred* array allows correct backtracking to retrieve the subsequence.

Time Complexity Analysis

The algorithm contains a nested loop where:

- The outer loop iterates over all n elements of A.
- For each $A[i]$, the inner loop iterates over $j = 1$ to $i - 1$.

Thus, the total number of comparisons is approximately

$$\sum_{i=1}^{n} (i - 1) = O(n^2).$$

Reconstruction of the subsequence requires at most $O(n)$ time. Overall, the algorithm runs in $O(n^2)$ time.

Problem 4: Removing the numbers optimally [20 pts]

Given a sequence of (positive) numbers, we want to remove the numbers from the sequence one by one. When removing one number x, we gain a score equal to l^2xr^2 where l is the number to the left of x in the current sequence ($l = 1$ if x is the leftmost number in the current sequence) and r is the number to the right of x in the current sequence ($r = 1$ if x is the rightmost number in the current sequence). For example, suppose the given sequence is $(2, 9, 12, 3)$. If we remove 12 first, the score we gain is $9^2 \times 12 \times 3^2 = 8748$, and the sequence becomes $(2, 9, 3)$. Now if we remove 9 from the sequence, then the score is $2^2 \times 9 \times 3^2 = 324$, and the sequence becomes $(2, 3)$. Next if we remove 3, then the score is $2^2 \times 3 \times 1^2 = 12$, and the sequence becomes (2). Finally we remove 2, and the score is $1^2 \times 2 \times 1^2 = 2$. The total score we gain is $8748 + 324 + 12 + 2 = 9086$. Our goal is to find an order to remove the numbers from the given sequence such that the total score we gain is maximized. Formally, design an algorithm $\textsc{Remove}(n, A)$ where A is an array of n positive numbers; the algorithm returns the maximum total score we can gain if the given sequence is A (for convenience, you are not required to return the optimal order for removing the numbers). Describe the basic idea of your algorithm and give the pseudocode. Briefly justify its correctness and analyze its time complexity. Your algorithm should run in $O(n^3)$ time or faster.

Solution.

Algorithm Idea

Extend the input array A by defining a new array B as follows:

$$B[0] = 1, \quad B[i] = A[i] \text{ for } 1 \le i \le n, \quad B[n+1] = 1.$$

Let $dp[i][j]$ denote the maximum score obtainable from removing all the numbers in the subarray $B[i+1 \dots j-1]$. If $B[k]$ is the *last* number removed in the interval (i, j), then its removal yields a score

$$B[i]^2 \times B[k] \times B[j]^2,$$

since $B[i]$ and $B[j]$ are then its neighbors. Therefore, the recurrence is:

$$dp[i][j] = \max_{i<k<j} \left\{ dp[i][k] + dp[k][j] + B[i]^2 \times B[k] \times B[j]^2 \right\},$$

with the base case $dp[i][i+1] = 0$ (no number to remove).

Pseudocode

```
1    FUNCTION Remove(n, A):
2        B[0] <- 1
```

```
        for i from 1 to n do:
            B[i] <- A[i]
        B[n+1] <- 1

        for i from 0 to n do:
            dp[i][i+1] <- 0

        for length from 2 to n+1 do:
            for i from 0 to n+1-length do:
                j <- i + length
                dp[i][j] <- 0
                for k from i+1 to j-1 do:
                    temp <- dp[i][k] + dp[k][j] + (B[i] * B[i]) * B[k] * (
                        ↪ B[j] * B[j])
                    if temp > dp[i][j] then:
                        dp[i][j] <- temp
                    end if
                end for
            end for
        end for

        return dp[0][n+1]
    END FUNCTION
```

Correctness Justification

- We define subproblems $dp[i][j]$ as the maximum score for removing all numbers between $B[i]$ and $B[j]$.
- By assuming $B[k]$ is the last to be removed in (i, j), its removal score is fixed as $B[i]^2 \times B[k] \times B[j]^2$, while the optimal scores for the remaining parts are given by $dp[i][k]$ and $dp[k][j]$.
- Maximizing over all choices of k ensures the optimal solution for each subinterval.

Time Complexity Analysis

There are $O(n^2)$ subproblems (indexed by pairs (i, j)), and for each subproblem we try $O(n)$ possible positions k. Hence, the overall time complexity is $O(n^3)$.

Problem 5: Tutorial of Fast Fourier Transform [10 pts]

This is not really a problem but more like a tutorial. Fast Fourier transform is arguably the most important algorithm in signal processing and has enormous applications in various fields (EE, CS, optics, acoustics, etc). Please watch the video from https://www.youtube.com/watch?v=spUNpyF58BY about Fourier transform. Even if you are familiar with the concept, the visualizations in the video are unparalleled. It's also highly recommended that you watch other videos from that content maker. Based on the video, answer the following questions:

(1) Continuous Fourier transform.

   (i) Considering Fourier transform as a function, what is its domain and range?

   (ii) Intuitively explain that if the rotation frequency matches some intrinsic frequency of the input signal, the "center of mass" will pop up.

   (iii) The examples in the video all have the "center of mass" on the x-axis. Should it always be the case?

(2) Discrete Fourier transform. In practice, we only have discrete samples of a signal (typically evenly spaced). Let $x_1, \ldots, x_N$ be the N sequential samples of a continuous signal with duration T. Write out the equation of the discrete Fourier transform that approximates the continuous version. What's the complexity to compute the transform for one frequency value?

(3) Discrete Fourier transform continued. Since we only have N samples, it is intuitive that the fastest frequency we can assess is N/T and the slowest is $1/T$. We, therefore, only care about the frequencies in the list $1/T, 2/T, 3/T, \ldots, N/T$. Rewrite the discrete transforms as a function of k, where the frequency is k/T (for simplicity, we refer to it as frequency k below). What's the complexity if we want to compute the transform for all N frequency values, i.e., we discretize the "center of mass" v.s. frequency figure shown in the video with N evenly-spaced frequency samples as well?

(4) Symmetry. Considering the previous discrete Fourier transform, compute the equation for frequency $k+N$ and compare it with the frequency k. You may need to use Euler's formula.

   We refer to the relationship as a "symmetry". The following is not directly related to the "symmetry", but can provide some useful intuitions.

   Now, think of the entire transformation into N frequencies as a matrix multiplication. I.e., we have N samples forming a vector as input, and we have N outputs for

the N frequencies. Try to write down a $N \times N$ matrix that performs the transformation.

(5) Fast Fourier Transform. For frequency k, separate the calculation into two parts, one part containing the x_n's where n is even and the other part containing the x_n's where n is odd. Now, consider the even (odd) part as a new transform problem with only half of the samples. Indeed, we are using the idea of **divde-and-conquer**, but how do we save computation? What about the symmetry above? Draft the algorithm to compute discrete Fourier transform in this way and analyze its complexity. For simplicity, you may assume N is a power of 2.

Solution.

1. **Continuous Fourier Transform.**

   (a) **Domain and Range.** The continuous Fourier transform of a signal

   $$f(t) \quad \text{(typically in } L^1(\mathbb{R}) \text{ or } L^2(\mathbb{R}))$$

   is defined as

   $$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(t)\, e^{-i\omega t} dt.$$

   Here, the transform is a function of the frequency ω. Thus, the domain is $\omega \in \mathbb{R}$ and the range is in $\mathbb{C}$ (i.e., complex numbers).

   (b) **Center of Mass Intuition.** Think of the Fourier transform as computing a weighted *center of mass* in the complex plane by summing rotated copies of the signal. When the rotation frequency ω in $e^{-i\omega t}$ coincides with an intrinsic (or dominant) frequency of the signal, the corresponding complex contributions tend to align. They add mostly in one direction rather than cancelling, so the "center of mass" (i.e., the average of the rotating contributions) will have a large magnitude.

   (c) **Is the Center of Mass Always on the x-axis?** Not necessarily. In the examples shown, the center of mass lie on the real axis (x-axis) because it only consider the real part of its fourier transform. In general, the computed value $\hat{f}(\omega)$ is a complex number and may lie anywhere in the complex plane.

2. **Discrete Fourier Transform (DFT):** In practice we have N evenly spaced samples $x_1, \ldots, x_N$ of a continuous signal of duration T. One common way to approximate the continuous Fourier transform is to write:

   $$X(\omega) \approx \Delta t \sum_{n=1}^{N} x_n e^{-i\omega t_n},$$

   where $\Delta t = T/N$ and $t_n = (n-1)\Delta t$. To compute the transform for one particular frequency value ω, one must sum over N terms, which takes $O(N)$ time.

3. **Discrete Fourier Transform with Frequency Index k.** Since the fastest frequency we can resolve is roughly N/T (and the slowest is $1/T$), we are interested in the discrete set of frequencies:

   $$\omega_k = \frac{k}{T}, \quad k = 1, 2, \ldots, N.$$

   Rewriting the transform, if we shift indices and (for convenience) let $n = 0, 1, \ldots, N-1$ (and similarly for k), then

   $$X[k] = \sum_{n=0}^{N-1} x[n]\, e^{-i2\pi \frac{k\,n}{N}}, \quad k = 0, 1, \ldots, N-1.$$

Computing one such $X[k]$ takes $O(N)$ time. Therefore, to compute all N outputs by evaluating the sum for each k, the complexity is $O(N^2)$.

4. **Symmetry.** First, consider the DFT formula:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, e^{-i2\pi \frac{k\,n}{N}}.$$

For frequency $k+N$, we have:

$$X[k+N] = \sum_{n=0}^{N-1} x[n]\, e^{-i2\pi \frac{(k+N)n}{N}} = \sum_{n=0}^{N-1} x[n]\, e^{-i2\pi \frac{k\,n}{N}}\, e^{-i2\pi n}.$$

Since $e^{-i2\pi n} = 1$ for any integer n, we conclude:

$$X[k+N] = X[k].$$

Hence, the DFT is periodic with period N.

Viewing the DFT as a matrix operation, define the $N \times N$ matrix F with entries:

$$F_{k,n} = e^{-i2\pi \frac{k\,n}{N}}, \quad k, n = 0, 1, \ldots, N-1.$$

Then, if $\mathbf{x} = [x[0], x[1], \ldots, x[N-1]]^T$, the DFT is given by:

$$\mathbf{X} = F\,\mathbf{x}.$$

5. **Fast Fourier Transform (FFT).** The key idea is to split the sum for each frequency into contributions from even and odd-indexed samples. For $k = 0, 1, \ldots, N-1$, write:

$$\begin{aligned} X[k] &= \sum_{n=0}^{N-1} x[n]\, e^{-i2\pi \frac{k\,n}{N}} \\ &= \sum_{m=0}^{\frac{N}{2}-1} x[2m]\, e^{-i2\pi \frac{k\,(2m)}{N}} + \sum_{m=0}^{\frac{N}{2}-1} x[2m+1]\, e^{-i2\pi \frac{k\,(2m+1)}{N}} \\ &= E[k] + e^{-i2\pi \frac{k}{N}}\, O[k], \end{aligned}$$

where

$$E[k] = \sum_{m=0}^{\frac{N}{2}-1} x[2m]\, e^{-i2\pi \frac{k\,m}{N/2}}, \quad O[k] = \sum_{m=0}^{\frac{N}{2}-1} x[2m+1]\, e^{-i2\pi \frac{k\,m}{N/2}}.$$

Notice that both $E[k]$ and $O[k]$ are DFTs of sequences of length $N/2$. By recursively computing these two DFTs and then combining them with the twiddle factors $e^{-i2\pi k/N}$, we avoid redundant computations.

Pseudocode:

```
FUNCTION FFT(x):
    N <- length(x)
    if N == 1 then
        return x
    end if
    x_even <- [ x[0], x[2], ..., x[N-2] ]
    x_odd  <- [ x[1], x[3], ..., x[N-1] ]

    E <- FFT(x_even)
    O <- FFT(x_odd)

    for k from 0 to N/2 - 1 do:
        t <- exp(-i * 2*pi*k/N) * O[k]
        X[k]       <- E[k] + t
        X[k+N/2]   <- E[k] - t
    end for
    return X
END FUNCTION
```

Since the recurrence splits the problem into two halves and does $O(N)$ work to combine the halves, the overall complexity of the FFT is $O(N \log N)$.

Problem 6: Find the common number [5 + 5 pts]

We are given a list of integers $x_1, \ldots, x_n$. We know that at least $\lceil n/2 \rceil$ of those numbers have the same value. Unfortunately, you don't have access to the list directly, and the only oracle you have is COMPARE(i, j), which tells you if x_i and x_j are equal or not.

(1) Design an oracle that finds the common number. The algorithm has to be deterministic. Analyze the correctness and the complexity in terms of the number of COMPARE.

(2) What if instead of $n/2$, we call any number that appears at least $\lceil n/k \rceil$ as a common number, where k is a constant integer with value ≥ 2. We aim to find all the common numbers. You can also directly solve this problem, which generalizes the previous one.

Solution. **Pseudocode:**

```
FUNCTION FindMajority(n):
    candidate_index <- 1
    count <- 1
    for i from 2 to n do:
        if compare(candidate_index, i) then
            count <- count + 1
        else
            count <- count - 1
            if count == 0 then
                candidate_index <- i
                count <- 1
            end if
        end if
    end for
    return candidate_index
END FUNCTION
```

Correctness:

- **Elimination:** Whenever two different numbers are encountered, they annihilate each other. Since a majority element appears in at least $\lceil n/2 \rceil$ positions, it is guaranteed that it is never completely eliminated.
- **Candidate:** After one pass, the remaining candidate is the majority element.

Complexity: The algorithm uses one pass over the list and performs one `compare` per iteration, giving a total of $O(n)$ comparisons.

(2)

When an element appears at least $\lceil n/k \rceil$ times (with $k \geq 2$) there can be at most $k-1$ such numbers. We use a *generalized Boyer–Moore algorithm* that maintains a set (or dictionary) of at most $k-1$ candidate elements with their counts.

Pseudocode:

```
FUNCTION FindAllCommon(n, k):
    Initialize D as an empty set (or dictionary)
    for i from 1 to n do:
        found <- False
        for each candidate in D do:
            if compare(candidate.index, i) then:
                candidate.count <- candidate.count + 1
                found <- True
                break out of inner loop
            end if
        end for

        if not found then:
            if size(D) < k - 1 then:
                Add candidate with index i and count 1 to D
            else:
                for each candidate in D do:
                    candidate.count <- candidate.count - 1
                end for
                Remove all candidates from D with count 0
            end if
        end if
    end for
END FUNCTION
```

Correctness:

- **Candidate Maintenance:** Any element that occurs at least $\lceil n/k \rceil$ times must survive the elimination process; since there can be at most $k-1$ such elements, maintaining $k-1$ candidates ensures that none of the common numbers are lost.
- **Verification:** A second pass verifies that each candidate really appears at least $\lceil n/k \rceil$ times.

Complexity:

- In the first pass, each new element is compared against at most $k-1$ candidates. With k a constant, this is $O(n)$ comparisons.

- The verification pass takes $O(n)$ comparisons per candidate, which is again $O(n)$ overall since there are at most $k-1$ candidates.

Thus, the total number of comparisons remains $O(n)$ when k is a constant.

Problem 7: Monitoring the statistics [5 pts]

A machine learning student wants to test empirically about sampling Gaussians. The student calls a standard Gaussian sampler at a time, which outputs a real value (e.g., drop one ball down along a very large Galton board). The student then performs sampling repeatedly and wants to keep track of the statistics.

Formally, the student gets x_t coming in a stream, which is generated from a Gaussian sampler. At every time point $t \geq 20$, the student wants to keep a record of three statistics: the 25, 50, and 75 percentiles of all the numbers sampled so far. More precisely, the 25 percentile at time t should return the data point x in $x_1, \ldots, x_t$ with rank $\lfloor 0.25 \times t \rfloor$ (similarly for 50 and 75). Suppose the student performs T sampling in total, what's the total complexity of your algorithm?

Solution. To compute the 25th, 50th, and 75th percentiles at every time point t (for $t \geq 20$) in a stream of Gaussian samples, we can use an augmented balanced binary search tree that supports the following operations:

(1) **Insert:** Insert a new sample x_t in $O(\log t)$ time.

(2) **Select:** Retrieve the element with a given rank (the $\lfloor 0.25 \times t \rfloor$, $\lfloor 0.50 \times t \rfloor$, and $\lfloor 0.75 \times t \rfloor$-th element) in $O(\log t)$ time.

Per time step cost: At each time t, we:

- Insert the new sample: $O(\log t)$.
- Query three order statistics (for 25th, 50th, and 75th percentiles): $3 \times O(\log t) = O(\log t)$ (since constants are ignored).

Thus, each time step takes $O(\log t)$ time. Over T total samples, the total cost is:

$$\sum_{t=1}^{T} O(\log t) = O(T \log T).$$

Conclusion: The total complexity for processing T samples and maintaining the required percentiles is $O(T \log T)$.

Problem 8: Multi-selection [5 pts]

We are given a list of integers $x_1, \ldots, x_n$ (not necessarily sorted). We are also given k distinct integer numbers $1 \leq i_1, \ldots, i_k \leq n$ (in sorted order), and we want to find the k elements of $x_1, \ldots, x_n$ that have ranks $i_1, \ldots, i_k$. Design the algorithm and analyze its correctness and complexity. Another special requirement here is that k is not a constant, so the complexity of k should also be considered.

Solution. **Idea:** We generalize the idea of Quickselect (or deterministic linear-time selection) and use it to find several order statistics simultaneously. Since the requested ranks are given in sorted order, we can use a divide-and-conquer approach:

- Pick one requested rank in the middle, say q^* (which is $i_{\lfloor k/2 \rfloor + 1}$).
- Use a linear-time selection algorithm to find the element p with global rank $r = q^*$.
- Partition the array around p.
- Split the query list into those that need to be found in the left partition (with ranks less than r) and the right partition (with ranks greater than r; adjusting the ranks by subtracting r).
- Recurse on each part.

Pseudocode:

```
FUNCTION MultiSelect(x, n, Q)
    // x[1..n] is the current list.
    // Q = {q_1, q_2, ..., q_k} is a sorted list of desired ranks.
    if Q is empty then
        return
    end if
    if n is small then
        sort x;
        return { x[q] for each q in Q }.
    end if
    let j = floor(|Q|/2) + 1;
    let q* = Q[j];   // this is the desired rank in x.
    p = Select(x, n, q*);
    (L, E, R) = Partition(x, p);
    r = |L| + 1;
    result = {}.
    if r is in Q then
        add p as the answer for rank r;
        remove r from Q.
    end if
```

```
        Q_left = { q in Q such that q < r }.
        Q_right = { q - r for q in Q such that q > r }.

        result_left = MultiSelect(L, |L|, Q_left);
        result_right = MultiSelect(R, |R|, Q_right);

        return union of result, result_left, and result_right.
    END FUNCTION
```

Correctness:

- The algorithm always finds an element p whose rank r matches one of the requested queries; the partitioning then correctly separates the remaining queries into those for the left and right parts.
- By induction on the subproblem sizes, the algorithm returns the element with the correct rank for every query.

Complexity: At every recursion level we:

- Spend $O(n)$ time to perform selection and partitioning.
- Split the k queries into two smaller sets.

Since the queries (of amount k) are split in half (roughly) at every partitioning step, the recursion will have $O(\log k)$ levels. (In the worst-case when $k = \Theta(n)$, the running time becomes $O(n \log n)$, which is the cost of sorting.) Thus, the overall time is $O(n \log k)$.