Assignment 2

1. [6 marks] **Reduce to known**. In the 2-Way-2- Sum decision problem, we are given arrays A and B of length n containing (not necessarily distinct) integers, and we must determine whether there are two pairs of indices $i_1, j_1, i_2, j_2 \in \{1, 2, \dots, n\}$ (not necessary distinct) for which

$$A[i_1] + A[i_2] = B[j_1] + B[j_2].$$

Design an efficient algorithm that solves the 2-Way-2- Sum problem and has time complexity $o(n^3)$ in the setting where operations on individual integers take constant time.

Note, that brute force solution that tries all possible quadruples of indices will have the running time of $\Theta(n^4)$. Less straightforward solution that reduces the problem to 3- Sum which is described in lecture notes, will have the running time of $\Theta(n^3)$. So both of these solutions are unacceptable and will receive 0 marks.

Main Idea

The brute force approach to checking all quadruples of indices requires Θ(n) time. Which can be reduced to 3-Sum which yields $\Theta(n^3)$. This would achieve $o(n^3)$, which we can then transform the problem into a set intersection problem: this would compute all possible pairwise sums from each array ($\Theta(n^2)$ sums each) and then check if any of the sum appears in both sets. Then using a hash-free approach with sorting and two-pointer technique, we can determine if the arrays share a common sum in $\Theta(n^2 \log n)$ time.

Algorithm

Pseudocode:

```
Algorithm 2-Way-2-Sum(A, B, n):
    // Computes all pairwise sums from array A
    S_A ← empty array of size n²
    index ← 0
    for i ← 1 to n do
        for j ← 1 to n do
            S_A[index] ← A[i] + A[j]
            index ← index + 1
```

```
// Computes all pairwise sums from array B
S_B ← empty array of size n^2
index ← 0
for i ← 1 to n do
    for j ← 1 to n do
        S_B[index] ← B[i] + B[j]
        index ← index + 1

// Sorts both arrays
Sort(S_A, n^2)
Sort(S_B, n^2)

// Use the two-pointer technique to find common element
i ← 0
j ← 0
while i < n^2 and j < n^2 do
    if S_A[i] = S_B[j] then
        return TRUE
    else if S_A[i] < S_B[j] then
        i ← i + 1
    else
        j ← j + 1

return FALSE
```

Correctness

Claim: The algorithm returns TRUE if and only if there exist indices i , i , j , j {1,2,...,n} such that A[i] + A[i] = B[j] + B[j].

Proof:

- **( )** If the algorithm returns TRUE, then at some point S_A[i] = S_B[j]. By construction of S_A and S_B in the first two loops, we have S_A[i] = A[i] + A[i] for some i , i and S_B[j] = B[j] + B[j] for some j , j . Therefore, A[i] + A[i] = B[j] + B[j].
- **( )** If there exist indices with A[i] + A[i] = B[j] + B[j], then this common sum appears in both S_A and S_B. After sorting, the two-pointer technique will find this common element: both pointers will eventually reach positions containing this value, and the algorithm will return TRUE.

The two-pointer technique correctly identifies common elements in sorted arrays by advancing the pointer pointing to the smaller value, ensuring all elements are considered.

Running Time Analysis

Time Complexity:

- Computing S_A: Two nested loops, each running n times → $\Theta(n^2)$
- Computing S_B: Two nested loops, each running n times → $\Theta(n^2)$
- Sorting S_A: Sorting n^2 elements → $\Theta(n^2 \log(n^2)) = \Theta(n^2 \log n)$
- Sorting S_B: Sorting n^2 elements → $\Theta(n^2 \log(n^2)) = \Theta(n^2 \log n)$
- Two-pointer search: Each pointer moves at most n^2 times → $\Theta(n^2)$

Total: $\Theta(n^2) + \Theta(n^2) + \Theta(n^2 \log n) + \Theta(n^2 \log n) + \Theta(n^2) = \Theta(n^2 \log n)$

Since $\Theta(n^2 \log n) = o(n^3)$, the algorithm satisfies the required time complexity.

Space Complexity: $\Theta(n^2)$ for storing the sums.

2. [6 marks] **Divide-and-Conquer/Binary Search.** Given a sorted in ascending order array $X[1..n]$ of distinct integers, the `CRS` operation was applied to it several times. The `CRS` (Circular Right Shift) operation shifts all entries 1 position to the right with the last element moved to the first entry:

```
temp := X[n]; for i from 1 to n-1 do X[i+1]:=X[i] od; X[1] := temp;
```

Give an algorithm with complexity in $o(n)$ to find the largest value in this array. Justify correctness and analyze running time. The number of times `CRS` operation was applied to the given array is unknown, but we know that array was sorted at the beginning. **NOTE!!!:** Your algorithm **MUST** have running time in $o(n)$, so simple scanning from left to right is not going to work and will receive 0 marks.

Main Idea

After k amount of applications of the CRS operation to a sorted array, there will exist exactly one position where the sorted order "breaks" — where a larger element is immediately followed by a smaller element. The maximum element is located at this break point. We can then locate this position using binary search by examining the relationship between elements at the midpoint and the boundaries, determining which half contains the break point. This yields O(log n) time complexity.

Algorithm

Pseudocode:

```
Algorithm FindMax(X, n):
    // Handle base case
    if n = 1 then
        return X[1]

    // Check if array is not rotated
    if X[1] < X[n] then
        return X[n]

    // Binary search for the break point
    left ← 1
    right ← n

    while left   right do
        // If we have only two elements
        if right - left = 1 then
            return max(X[left], X[right])

        mid ← floor((left + right) / 2)

        // Check if mid is the maximum
        if X[mid] > X[mid + 1] then
            return X[mid]

        // Check if mid - 1 is the maximum
        if X[mid] < X[mid - 1] then
            return X[mid - 1]

        // Decide which half to search
        if X[mid] > X[right] then
            // Break point is in right half
            left ← mid + 1
        else
            // Break point is in left half
            right ← mid - 1

    return X[left]
```

Correctness

Loop Invariant: At the start of each iteration, the maximum element lies within the range [left, right].

Initialization: Initially left = 1 and right = n, so the maximum is in [1, n].

Maintenance: We show the invariant is preserved in each case:

- **Case 1:** If X[mid] $>$ X[mid+1], then mid is the break point and X[mid] is the maximum.
- **Case 2:** If X[mid] $<$ X[mid-1], then mid-1 is the break point and X[mid-1] is the maximum.
- **Case 3:** If X[mid] $>$ X[right], the portion [left, mid] is not entirely sorted in ascending order (since X[mid] $>$ X[right] contradicts global ascending order), so the break point must be in (mid, right]. We set left $\leftarrow$ mid + 1.
- **Case 4:** If X[mid]   X[right], the portion [mid, right] is sorted in ascending order, so the break point must be in [left, mid-1]. We set right $\leftarrow$ mid - 1.

Termination: The loop will terminate when we find the maximum (Cases 1, 2) or when the range contains   2 elements, at which point we return the larger.

Example: Consider array [5, 6, 7, 1, 2, 3, 4] (originally [1,2,3,4,5,6,7] with 3 CRS operations).

- Iteration 1: left=1, right=7, mid=4, X[4]=1, X[7]=4. Since X[mid]=1 $<$ X[right]=4, set right=3.
- Iteration 2: left=1, right=3, mid=2, X[2]=6, X[3]=7. Since X[mid]=6 $<$ X[mid+1]=7 and X[mid]=6 $>$ X[right]=7 is false, set left=3.
- Iteration 3: left=3, right=3. Check: X[3]=7 $>$ X[4]=1, return 7.

Running Time Analysis

The algorithm performs binary search:

- Each iteration reduces the search space by approximately half
- Number of iterations: at most log n + 1
- Work per iteration: O(1) comparisons and assignments

Total Time Complexity: O(log n) = o(n)

Space Complexity: O(1)

3. [6 marks] (a) [4 marks] Solve the following recurrence by the recursion-tree method (you may assume that n is a power of 2):

$$T(n) = \begin{cases} 7, & n = 1, \\ 4T(n/2) + 3 \cdot n^2, & n > 1. \end{cases}$$

(b) [2 marks] Solve part (a) using Master theorem.

Part (a): Recursion Tree Method [4 marks]

Given: T(n) = 4T(n/2) + 3n^2 for n > 1, and T(1) = 7

Solution: First we construct a recursion tree where each node represents a subproblem:

Level 0 (root):

- 1 node of size n
- Work: 3n^2

Level 1:

- 4 nodes, each of size n/2
- Work per node: 3(n/2)^2 = 3n^2/4
- Total work: 4 × 3n^2/4 = 3n^2

Level 2:

- 4^2 = 16 nodes, each of size n/4
- Work per node: 3(n/4)^2 = 3n^2/16
- Total work: 16 × 3n^2/16 = 3n^2

Level i (general):

- 4^i nodes, each of size n/2^i
- Work per node: 3(n/2^i)^2 = 3n^2/4^i
- Total work: 4^i × 3n^2/4^i = 3n^2

Tree Height: The recursion stops when n/2^h = 1, i.e., h = log n **Leaf Level:**

- Number of leaves: 4^(log n) = n^(log 4) = n^2
- Work per leaf: T(1) = 7
- Total work at leaves: 7n^2

Summing all levels: T(n) = (work at internal levels) + (work at leaves) T(n) = 3n^2 × log n + 7n^2 T(n) = 3n^2 log n + 7n^2

Therefore: T(n) = Θ(n^2 log n)

Part (b): Master Theorem [2 marks]

Given: T(n) = 4T(n/2) + $3n^2$

Master Theorem form: T(n) = aT(n/b) + f(n) where a   1, b > 1

Identify parameters:

- a = 4 (number of recursive calls)
- b = 2 (subproblem size divisor)
- f(n) = $3n^2$ (non-recursive work)

Step 1: Compute n^(log_b a) n^(log_b a) = n^(log 4) = n^2

Step 2: Compare f(n) with n^(log_b a) f(n) = $3n^2$ = Θ(n^2) = Θ(n^(log 4))

Step 3: Apply Master Theorem

This matches **Case 2** of the Master Theorem: If f(n) = Θ(n^(log_b a) log^k n) where k   0, then T(n) = Θ(n^(log_b a) log^(k+1) n) Here k = 0, so: T(n) = Θ(n^2 log n)

Both methods give: T(n) = Θ(n^2 log n)

4. [8 marks] **Divide-and-conquer** The input for this problem consists of n stations $s_1, \dots, s_n$ where station i ($i = 1, 2, \dots, n$) is given by its integer coordinates x_i and y_i in the plane. We say that station i can transmit to station j if station j is south-east of station i, i.e., $x_i < x_j$ and $y_i > y_j$. The ***load factor*** of station i is defined to be the number of stations it can transmit to. The goal is to find a station with largest load factor. If there are several stations with the same largest load factor - return the list of those.
Give a divide and conquer algorithm for this problem. You algorithm has to have worst case running time in $o(n^2)$ (i.e. it must be asymptotically faster than quadratic time direct algorithm).

Main Idea

The key insight is that after sorting stations by x-coordinate, the constraint x < x is automatically satisfied for all pairs where i < j in the sorted order. The problem is then reduced to counting, for each station i, how many stations j > i have y < y . This is equivalent to counting inversions in the y-coordinate sequence. We then use a divide-and-conquer approach based on merge sort that counts inversions while sorting, achieving O(n log n) time complexity.

Algorithm

Pseudocode:

```
Algorithm MaxLoadFactor(stations, n):
    // Sort stations by x-coordinate (maintain original indices)
    sortedStations ← Sort stations by x  in ascending order

    // Extract y-coordinates and create index mapping
    Y ← array of size n
    indices ← array of size n
    for i ← 1 to n do
        Y[i] ← sortedStations[i].y
        indices[i] ← sortedStations[i].originalIndex

    // Initialize load factors
    loadFactors ← array of size n, initialized to 0
    // Count inversions using divide-and-conquer
    MergeSortAndCount(Y, indices, loadFactors, 1, n)

    // Find maximum load factor
    maxLoad ← loadFactors[1]
    for i ← 2 to n do
        if loadFactors[i] > maxLoad then
            maxLoad ← loadFactors[i]

    // Collect all stations with maximum load factor
    result ← empty list
    for i ← 1 to n do
        if loadFactors[i] = maxLoad then
            result.append(i)

    return result

Algorithm MergeSortAndCount(Y, indices, loadFactors, left, right):
    if left   right then
        return

    mid ← floor((left + right) / 2)

    // Recursively sort and count in both halves
    MergeSortAndCount(Y, indices, loadFactors, left, mid)
    MergeSortAndCount(Y, indices, loadFactors, mid + 1, right)

    // Merge and count cross-inversions
    MergeAndCount(Y, indices, loadFactors, left, mid, right)
```

```
Algorithm MergeAndCount(Y, indices, loadFactors, left, mid, right):
    // Create temporary arrays
    tempY ← array of size (right - left + 1)
    tempIndices ← array of size (right - left + 1)

    i ← left          // Index for left subarray
    j ← mid + 1       // Index for right subarray
    k ← 0             // Index for temporary array

    // Merge while counting inversions
    while i   mid and j   right do
        if Y[i] > Y[j] then
            // Y[i] forms an inversion with all elements from j to right
            loadFactors[indices[i]] ← loadFactors[indices[i]] + (right - j + 1)
            tempY[k] ← Y[i]
            tempIndices[k] ← indices[i]
            i ← i + 1
        else
            tempY[k] ← Y[j]
            tempIndices[k] ← indices[j]
            j ← j + 1
        k ← k + 1

    // Copy remaining elements from left subarray
    while i   mid do
        tempY[k] ← Y[i]
        tempIndices[k] ← indices[i]
        i ← i + 1
        k ← k + 1

    // Copy remaining elements from right subarray
    while j   right do
        tempY[k] ← Y[j]
        tempIndices[k] ← indices[j]
        j ← j + 1
        k ← k + 1

    // Copy temporary arrays back to original
    for k ← 0 to (right - left) do
        Y[left + k] ← tempY[k]
        indices[left + k] ← tempIndices[k]
```

Correctness

Claim: The algorithm correctly computes the load factor for each station and identifies all stations with maximum load factor.

Proof:

Lemma 1: After sorting by x-coordinate, for any positions i < j in the sorted array, we have x_sorted[i] < x_sorted[j].

Proof of Lemma 1: This follows directly from the sorting step.

Lemma 2: Station at position i can transmit to station at position j (where i < j after sorting) if and only if Y[i] > Y[j].

Proof of Lemma 2: By Lemma 1, x < x is satisfied. The transmission condition requires additionally that y > y , which is exactly Y[i] > Y[j].

Lemma 3: The merge sort procedure correctly counts all inversions.

Proof of Lemma 3: We prove by induction on the size of the array.

- **Base case:** For arrays of size 1, there are no inversions. The algorithm correctly returns without counting.
- **Inductive step:** Assume the algorithm works correctly for arrays of size < n. For an array of size n:
- Inversions are partitioned into three disjoint sets:
    1. Inversions within the left half (counted by recursive call)
    2. Inversions within the right half (counted by recursive call)
    3. Cross-inversions where i is in the left half and j is in the right half
- During merge, when Y[i] > Y[j] where i is from the left half and j from the right half, Y[i] is greater than Y[j] and all subsequent elements in the right half (since the right half is sorted). We correctly add (right - j + 1) to the load factor.
- All three types are counted exactly once.

Main Correctness: By Lemmas 1-3, the load factor for each station equals the number of inversions it participates in, which equals the number of stations it can transmit to. The final scan correctly identifies all stations with maximum load factor.

Running Time Analysis

Line-by-line analysis:

1. **Sorting by x-coordinate:** O(n log n) using merge sort or any efficient comparison-based sort

2. **Extracting y-coordinates and indices:** Θ(n) — single loop through n stations

3. **MergeSortAndCount recursive analysis:**

- Recurrence relation: T(n) = 2T(n/2) + Θ(n)
- The merge step processes each element exactly once: Θ(n)
- By Master Theorem (a=2, b=2, f(n)=Θ(n)):
  - n^(log 2) = n
  - f(n) = Θ(n) = Θ(n^(log 2))
  - Case 2: T(n) = Θ(n log n)

4. **Finding maximum and collecting results:** Θ(n) — two linear scans

Total Time Complexity: T(n) = O(n log n) + Θ(n) + Θ(n log n) + Θ(n) = O(n log n)

Since O(n log n) = o(n^2), the algorithm satisfies the required time complexity.

Space Complexity: Θ(n) for temporary arrays during merging.

5. [4 marks] Consider the following recursive algorithm prototype in pseudocode:

```
int Fiction( A:: array, n:: integer) {
 if (n>1) {
  B <- A[1],...,A[n/3];         // copy 1st "third" of A to B
  C <- A[n/3+1],...,A[2*n/3];   // copy 2nd "third" of A to C
  D <- A[2*n/3+1],...,A[n];     // copy 3rd "third" of A to D
  C <- Perturb(C);
  cond2 <- Fiction(C, n/3);
  cond3 <- Fiction(D, n/3);
  if (cond2) {
    cond1 <- Fiction(B, n/3);
    cond2 <- (cond1 + cond2)/2;
  }
  return cond2;
 }
 else
  if A[1]>0 then return 1 else return 0;
}
```

(a) [3 marks] What is the worst case complexity of this algorithm assuming that the algorithm `Perturb` when applied to an array of length n has running time complexity $\Theta(n)$? To justify your answer provide and solve a divide-and-conquer recurrence for this algorithm. You may assume that n is a power of 3.

(b) [1 marks] What is the “best case” complexity of this algorithm under the same assumptions as in (a)?

Part (a): Worst Case Complexity [3 marks]

Analysis: The worst case occurs when the condition `if (cond2)` evaluates to TRUE at every recursive level, causing all three recursive calls to execute.

Operations at each level (worst case):

1. Copying three subarrays B, C, D: Each is size n/3, total Θ(n)
2. Perturb(C): Given as Θ(n/3) = Θ(n)
3. Three recursive calls: Fiction(C, n/3), Fiction(D, n/3), Fiction(B, n/3)
4. Arithmetic operations and comparisons: O(1)

Recurrence Relation:

T_worst(n) = 3T_worst(n/3) + Θ(n) for n > 1 T_worst(1) = Θ(1)

Solution by Master Theorem:

Parameters:

- a = 3 (three recursive calls)
- b = 3 (subproblem size factor)
- f(n) = Θ(n) (work per level)

Step 1: Compute n^(log_b a)
n^(log_b a) = n^(log 3) = n^1 = n

Step 2: Compare f(n) with n^(log_b a)
f(n) = Θ(n) = Θ(n^(log 3))

Step 3: Apply Master Theorem Case 2

Since f(n) = Θ(n^(log_b a) log^k n) where k = 0: T(n) = Θ(n^(log_b a) log n) = Θ(n log n)

Alternative verification by recursion tree:

- Level 0: cn work
- Level 1: 3 subproblems of size n/3, each doing c(n/3) work → total cn
- Level i: 3^i subproblems of size n/3^i, each doing c(n/3^i) work → total cn
- Height: log n
- Total: cn × log n = Θ(n log n)

Worst Case Time Complexity: Θ(n log n)

Part (b): Best Case Complexity [1 mark]

Analysis: The best case occurs when `if (cond2)` always evaluates to FALSE, causing the recursive call Fiction(B, n/3) to be skipped.

Operations at each level (best case):

1. Copying three subarrays: Θ(n)
2. Perturb(C): Θ(n)
3. Only two recursive calls: Fiction(C, n/3) and Fiction(D, n/3)
4. The third call Fiction(B, n/3) is skipped

Recurrence Relation:

T_best(n) = 2T_best(n/3) + Θ(n) for n > 1 T_best(1) = Θ(1)

Solution by Master Theorem:

Parameters:

- a = 2 (two recursive calls)
- b = 3 (subproblem size factor)
- f(n) = Θ(n)

Step 1: Compute n^(log_b a) n^(log_b a) = n^(log 2)   n^0.631

Step 2: Compare f(n) with n^(log_b a) f(n) = Θ(n) = n^1 n^(log 2) n^0.631

Since 1 > 0.631, we have f(n) = Ω(n^(log 2 +)) for   = 0.2 (for example)

Step 3: Verify regularity condition for Case 3

Need: af(n/b)   cf(n) for some constant c < 1 2 · (n/3) = 2n/3   (2/3) · n Taking c = 2/3 < 1, the condition is satisfied.

Step 4: Apply Master Theorem Case 3

T(n) = Θ(f(n)) = Θ(n)

Best Case Time Complexity: Θ(n)