- **Question 1-1:** Array of matching pairs
  - Instead of sorting both arrays independently, quick-sort partitioning to match books to covers.
    - 1: Choose a random book as the pivot
    - 2: Partition all the covers into: Covers that are smaller than the book, covers that are equal to the book (this is the match), and covers that are larger than the book
    - 3: Partition the book in the same way: Books smaller than cover, book ,equal to cover(Match), and books larger than the cover
    - 4: Then recursive apply this partitioning to the left and right subarrays
    - 5: When an array has one element, it is already matched
  - Time Complexity:
    - Partitioning both arrays takes O(n).
    - Each recursive call reduces the problem size by half which is O (log n) levels of recursion.
    - Overall Complexity: O(n log n)
  - Hashtable approach: Hash table stores the books identifiers, and then iterate through the covers and check if the book exists within the hash table
    - O(n) (BEST SOLUTION)
- **Question 1-2:** (O(n)) approach
  - Traverses the array one time and swaps the elements as needed
    - If even: indices(i), ensure arr[i] <= arr[i+1]
    - If odd: indices(i), ensure arr[i] >= arr[i+1]
    - Swap when needed, that leads to O(n) time
  - Time complexity:
    - Loop runs O(n) times (One pass through the array).
    - Swaps occur at most O(n) times, each in constant O(1) time
    - Total Complexity: O(n)
- **Question 2 is attached separately**
- **Question 3-1: Insertion sort on small arrays in merge sort**
  - A. Insertion sort has a worst-case time complexity of O(*k*^2) when sorting an array that is *k* length. Since there is *n / k* sublists the total time to sort all of these would be:
    - *(n / k) * O(k^2) = O(nk)*
    - Which means that sorting all these sublists using insertion sort would take *O(nk)* time
  - B. After sorting the *n / k* sublists, they need to be merged using the standard merge operation from merge sort
    - Preform a multiway merge of *n / k* sorted sublists of length *k*
    - Then a *k*-way merge happens using a priority queue (min-heap) takes O(*log(n / k))*
    - Which means the worst case time complexity of merging is: *O(n log(n / k))*
  - C. The total runtime of the modified merge sort is : *O(nk + n log(n / k))*
    - Set: *O(nk + n log(n / k)) = O(n log n)*

- Divide n: $O(k + \log(n / k)) = O(\log n)$
- Rearrange: $O(k) = O(\log n - \log(n / k))$

  $O(k) = O(\log k)$

  $k = O(\log n)$

This means the largest value of K allows the modified merge sort to maintain the same asymptotic complexity as standard merge sort = $O(\log n)$.