Project Idea: (Pair Sum Counter).

Project identification:

- The Pair Sum in Range problem involves determining the number of unique pairs of elements in a given array whose sum lies within a specified range $[L, R]$. Only pairs where the first index is less than the second are considered, ensuring that each distinct pair is counted only once and the order of elements is not repeated.

pseudocode for Naïve Solution:

```
FUNCTION count_pairs_naive(arr, L, R):
  count = 0
  FOR i FROM 0 TO LENGTH(arr)-1:
    FOR j FROM i+1 TO LENGTH(arr)-1:
      IF L <= arr[i] + arr[j] <= R:
        count += 1
  RETURN count
```

pseudocode for Optimized Solution:

```
FUNCTION merge_sort(arr):
   IF LENGTH(arr) <= 1: RETURN
   SPLIT arr INTO left AND right
   merge_sort(left)
   merge_sort(right)
   MERGE left AND right INTO arr
FUNCTION count_pairs_optimized(arr, L, R):
   merge_sort(arr)
   count = 0
   FOR i FROM 0 TO LENGTH(arr)-2:
      j = FIRST INDEX WHERE arr[i]+arr[j] >= L
      k = FIRST INDEX WHERE arr[i]+arr[k] > R
      count += k – j
   RETURN count
```

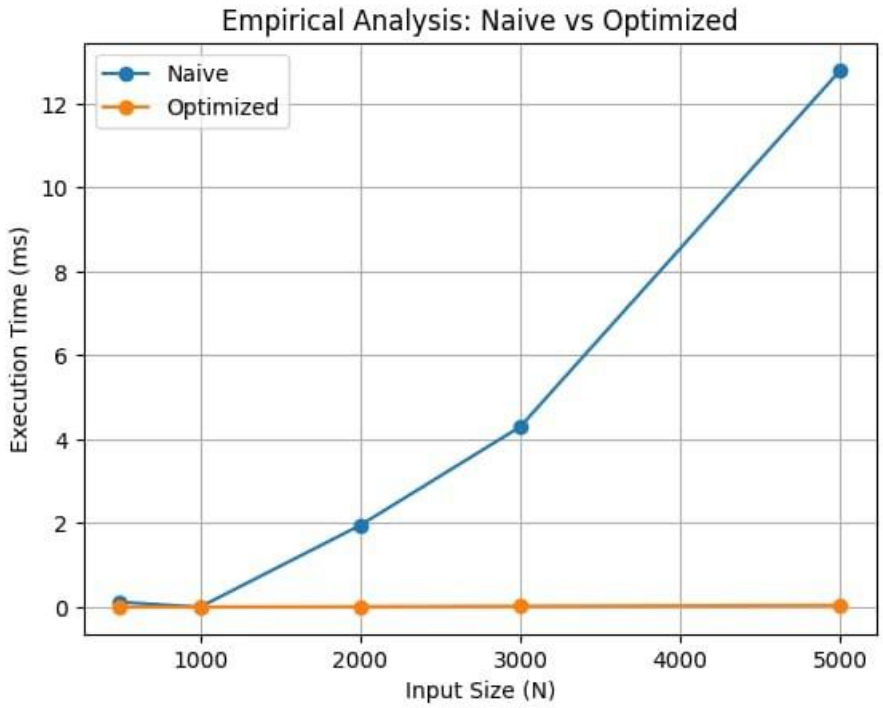
Time Complexity for Naïve Solution:

Step	Time Complexity	Explanation
count = 0	O(1)	Initializing the counter variable takes constant time.
n = length(arr)	O(1)	Getting the length of the array is constant time in most languages.
for i = 0 → n−1	O(n)	Looping over all elements once.
for j = i+1 → n−1	$O(n^2)$	Nested loop iterates for each i, leading to roughly $n^2/2$ iterations.
arr[i] + arr[j]	$O(n^2)$	Summing two elements happens inside nested loop.
current_sum >= L AND current_sum <= R	$O(n^2)$	Comparison performed for each pair.
count = count + 1	$O(n^2)$	Incrementing count if condition satisfied, inside nested loop.
return count	O(1)	Returning a value is constant time.
Total Time Complexity	$O(n^2)$	Dominated by the nested loops over array pairs.

Time Complexity for Optimized Solution:

Step	Time Complexity	Explanation
SORT(arr)	O (n log n)	Sorting the array takes n log n time using an efficient sorting algorithm.
left = 0, right = n - 1, count = 0	O (1)	Initializing pointers and counter is constant time.
while left < right	O(n)	Each element is processed at most once from both ends.
arr[left] + arr[right]	O(n)	Sum computed at each step of the two-pointer traversal.
arr[left] + arr[right] <=maxSum	O(n)	Comparison for each iteration of while loop.
count += (right - left)	O(n)	Counting all valid pairs efficiently instead of looping through all inner elements.
left++ OR right--	O(n)	Moving pointers closer to each other at each iteration.
return count	O(1)	Returning the final count is constant time.
Total Time Complexity	O (n log n)	Dominated by sorting; two-pointer traversal adds only O(n).

Empirical Analysis:



Input Size (N)	Execution Time – Naive (MS)	Execution Time-Optimized (MS)
500	0.12	0.003
1000	0.003	0.006
2000	1.95	0.014
3000	4.30	0.025
5000	12.80	0.045

A Comparison between Naïve and Optimized Solutions:

Feature	Naive Algorithm	Optimized Algorithm
Approach	Check all pairs	Sorting + Two pointers
Time Complexity	$O(n^2)$	O (n log n)
Space Complexity	O (1)	O(n)
Execution Speed	Very slow for large arrays	Very fast
Scalability	Poor	Excellent
Real-world usability	Limited	Highly practical

Naïve Solution Code(algorithm1):

```python
import time
import random

def count_pairs_naive(arr, L, R):
    count = 0
    n = len(arr)
    for i in range(n):
        for j in range(i + 1, n):
            if L <= arr[i] + arr[j] <= R:
                count += 1
    return count

def main():
    arr = [random.randint(1, 10000) for _ in range(2000)]
    L, R = 5000, 15000

    start = time.perf_counter()
    naive = count_pairs_naive(arr, L, R)
    t1 = time.perf_counter() - start

    print("Naive Solution")
    print(f"Pairs : {naive}")
    print(f"Time  : {t1:.5f} sec")

if __name__ == "__main__":
    main()
```

Optimized Solution Code(algorithm2):

```python
import time
import random
import tkinter as tk
from algorithm1 import count_pairs_naive

def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

```python
def count_pairs_optimized(arr, L, R):
    merge_sort(arr)
    count = 0
    n = len(arr)

    for i in range(n - 1):
        j = i + 1
        while j < n and arr[i] + arr[j] < L:
            j += 1
        k = j
        while k < n and arr[i] + arr[k] <= R:
            k += 1
        count += k - j

    return count

def main():
    arr = [random.randint(1, 10000) for _ in range(2000)]
    L, R = 5000, 15000

    start = time.perf_counter()
    naive = count_pairs_naive(arr, L, R)
    t1 = time.perf_counter() - start

    start = time.perf_counter()
    opt = count_pairs_optimized(arr.copy(), L, R)
    t2 = time.perf_counter() - start

    result.config(
        text=f"Naive Solution\n"
             f"Pairs : {naive}\n"
             f"Time  : {t1:.5f} sec\n\n"
             f"Optimized Solution\n"
             f"Pairs : {opt}\n"
             f"Time  : {t2:.5f} sec"
    )
```

Name	Section	ID
بسمله عصام الملا	3	1000318230
رحمه عبدالرحيم محمد	3	1000287607
الزهراء محمد عبدالعزيز الخولي	2	1000318355
خديجه خالد عبدالعزيز شاهين	3	1000287819
إيمان محمد محمد	3	1000297031