1. Develop an improved implementation of insertion sort for integer vector (insertion_sort_im) that precomputes the length of each vector before the sorting. Keep in mind that the vectors are sorted according to their length (see ivector_length function). You can test the correctness of your sorting algorithm using the provided check_sorted function.

I have replaced the naive function as "insertion_sort_original". The below function is the improvised version of insertion sort; As per my tests, the improvised version takes very less time than the naïve algorithm.

```
void insertion_sort(int** A, int n, int l, int r)
{
  int i;
  int key;
  int* vect_len;
  int arr[r];
  for (int k = 0; k <= r; k++)
          {
          arr[k] = ivector_length(A[k], n);
          }

  for (int j = l+1; j <= r; j++)
    {
      key = arr[j];
      vect_len = A[j];
      i = j - 1;

      while ((i >= l) && arr[i] > key)
        {
            arr[i+1] = arr[i];
            A[i+1] = A[i];
            i = i - 1;
          }

      arr[i+1] = key;
      A[i+1] = vect_len;
    }
}
```

2. Implement a merge sort for an array of integer vectors. As for the improved insertion sort algorithm, you should precompute the length of the vectors before the sorting and the sorting is done according to the vector length. Test the correctness of your merge sort implementation using the provided check_sorted function.

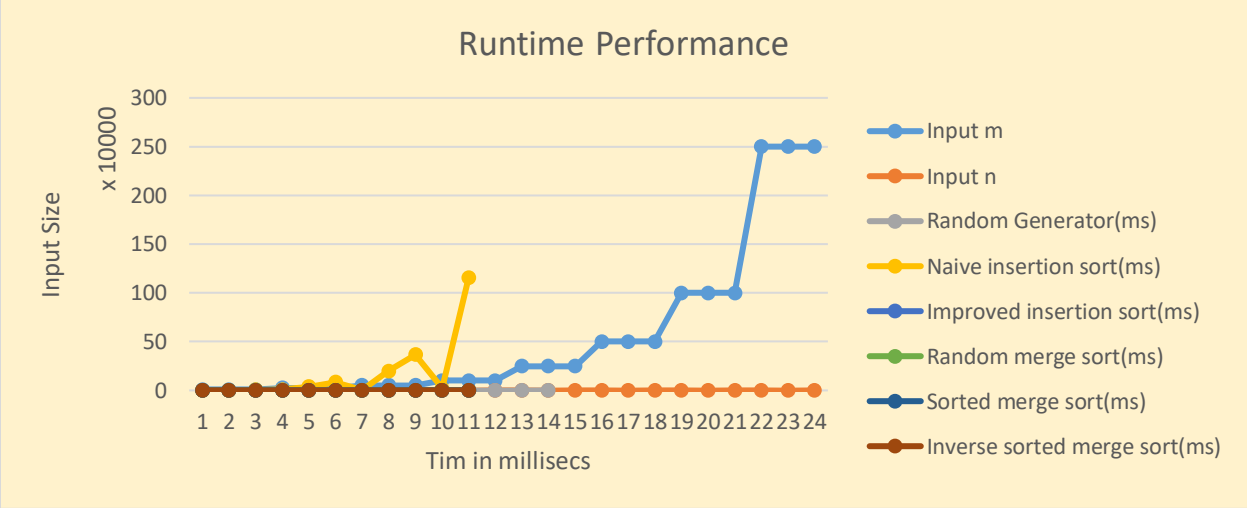
The below algorithm is for sorting vector elements using merge sort. The merge sort is very fast compared to insertion sort.

```
void merge(int** A, int p, int q, int r)
{
   int i, j, k;
   int l, n;
   int* vector_len;
   int array[r];
   for (int l = p; l <= r; l++)
         {
         array[l] = ivector_length(A[l], n);
         }
   i=p;
   j=q+1;
   k=p;

   int n1 = q - p + 1;
   int n2 =  r - q;

   int L[n1];
   int R[n2];

   for (i = 1; i <= n1; i++)
      L[i] = array[p+i];
   for (j = 1; j <= n2; j++)
      R[j] = array[q + 1 + j];
   i = 1;
   j = 1;
      while(i<=q &&j<=r)
         {
         if (L[i] <= R[j])
         {
                  array[k] = L[i];
                  i++;
         }
         else
         {
                  array[k] = R[j];
                  j++;
         }
```

```
        k++;
        }
        while(i<=q)
        {
        array[k] = L[i];
        k++;
        i++;
        }
}

void mergeSort(int** A, int p, int r)
{
   if (p < r)
   {
      int q = p+r/2;
      mergeSort(A, p, q);
      mergeSort(A, q+1, r);
      merge(A, p, q, r);
   }
}
```

3. Measure the runtime performance of insertion sort (naive and improved) and merge sort for random, sorted, and inverse sorted inputs of size m = 10000; 25000; 50000; 100000; 250000; 500000; 1000000; 2500000 and vector dimension n = 10; 25; 50. You can use the provided functions create_random_ivector, create_sorted_ivector, create reverse_sorted_ivector. Repeat each test a number of times (usually at least 10 times) and compute the average running time for each combination of algorithm, input, size m, and vector dimension n. Report and comment on your results.

The runtime performance varies for each algorithm and that is depicted in the Line chart below;



It was taking a long time for input sizes above m=100000 and n=25. The result set is provided below in a table;

Input m	Input n	Random Generator(ms)	Naive insertion sort(ms)	Improved insertion sort(ms)	Random merge sort(ms)	Sorted merge sort(ms)	Inverse sorted merge sort(ms)
10000	10	3	1398	1	0	0	0
10000	25	5	3678	1	0	0	0
10000	50	14	7157	2	0	0	0
25000	10	10	9553	1	0	0	0
25000	25	13	36951	4	0	0	0
25000	50	21	84090	7	0	0	0
50000	10	11	3837	3	0	0	0
50000	25	30	200730	10	0	0	0
50000	50	61	366658	13	0	0	0
100000	10	24	21237	8	0	0	0
100000	25	40	1154030	24	0	0	0
100000	50	90					
250000	10	64					
250000	25	164					