# SOFTWARE PROGRAMMING FOR PERFORMANCE

# ASSIGNMENT 1

**SUBMITTED BY -2019121001, 2018111040**

## **Question 1.** MATRIX MULTIPLICATION

The matrix multiplication is a trivial mathematical problem. Two 2D matrices with dimensions (p,q) and (q,r). The output matrix with dimensions with (p,r).

Optimizations Attempted :

1) **Tiling the matrix**: The first attempt was made to tile the matrix with a tile size of 64*64 to reduce the number of caches misses but due to the limited size of the array instead the running time got increased. The running time got increased from 3.6 to 4 sec. SO didn't use it.
2) **Converting all variables to register variables**: All the variables and pointer being used were converted to register variables and pointer to reduce the fetching time of the variables. The fetching time got reduced by 0.3 sec.
3) **Caching the most commonly used variables:** The part of the matrix that was being called again and again and the array being used again and again was stored in the register variable to reduce the running time. This reduced the running time drastically.

4) **Accessing the variables using pointers**: The arrays being used in the inner loop are accessed using the pointer and incremented to give better access time. This improved the running time drastically.
5) **Loop ordering**: The loop ordering improves the caching of the matrix due to which cache misses are reduced highly and hence running time improves. This improved the running time by 0.5 sec.
6) **Loop Unrolling**: The loop is unrolled such that it improves the performance since the ability to execute instructions becomes faster. This improved the running time by 0.6 sec.

After all the optimizations performed the running time reduced to 0.8 sec (vary from machine to machine and state of the machine). The best time recorded was 0.6 sec.

The code -

```
Matrix *matrix_multiply(Matrix *a,Matrix *b,int p,int
q,int r)
{
    Matrix *result;
    result = malloc(sizeof(Matrix));

    register int j = 0;
    register int i = 0;
    register int k = 0;
    register int df;
    register int *result_point = result->matrix[0];
    memset(result_point,0,p*r*sizeof(int));

    do
```

```
{
    i = 0;
    do
    {
        register int *b_sub = b->matrix[i];
        result_point = result->matrix[j];
        //va = _mm256_loadu_ps(a->matrix+(i*p)+j);
        df = a->matrix[j][i];
        do
        {
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            *result_point++ += df*(*b_sub++);
            k += 20;
        }while(k < r-19);

        while(k < r){
            *result_point++ += df*(*b_sub++);
```

```
                k++;
            }

            i++;
        }while(i < q);

        j++;

    }while(j < p);

    return result;
}
```

## Question 2. MERGE SORT

**Completely naive merge sort :**

**Reverse input**

1 . **Different Runnings** :

1. Took 0.275009 seconds to execute
2. Took 0.306700 seconds to execute
3. Took 0.296584 seconds to execute

2. **Optimisations Attempted** :

1. Merge Sort with Insertion sort for input below a certain number :
    a. Didn't make too much of a difference as it was not the bottleneck parameter.
2. Merge Sort with Iterative Character instead of Recursive calls :C

    a. Code became considerably faster as the function calls were reduced and a lot of cycles were saved
3. Merge Sort with Iterative Character instead of Recursive calls coupled with insertion sort for smaller portions :
    a. Code speed up keep increasing as merge sort is slower than insertion sort for smaller inputs
        i. Important fact : The speed up seems to be even more when we use powers of 2 for the size of the tiles for insertion sort
4. Bit hacks were applied and some for loops were converted to while loops to successfully speed up the code even further
5. Data prefetching instructions were used to prefetch and retain the data in the highest possible cache levels : This led to a further speed up of the code time because cache misses were reduced to an absolute minimum

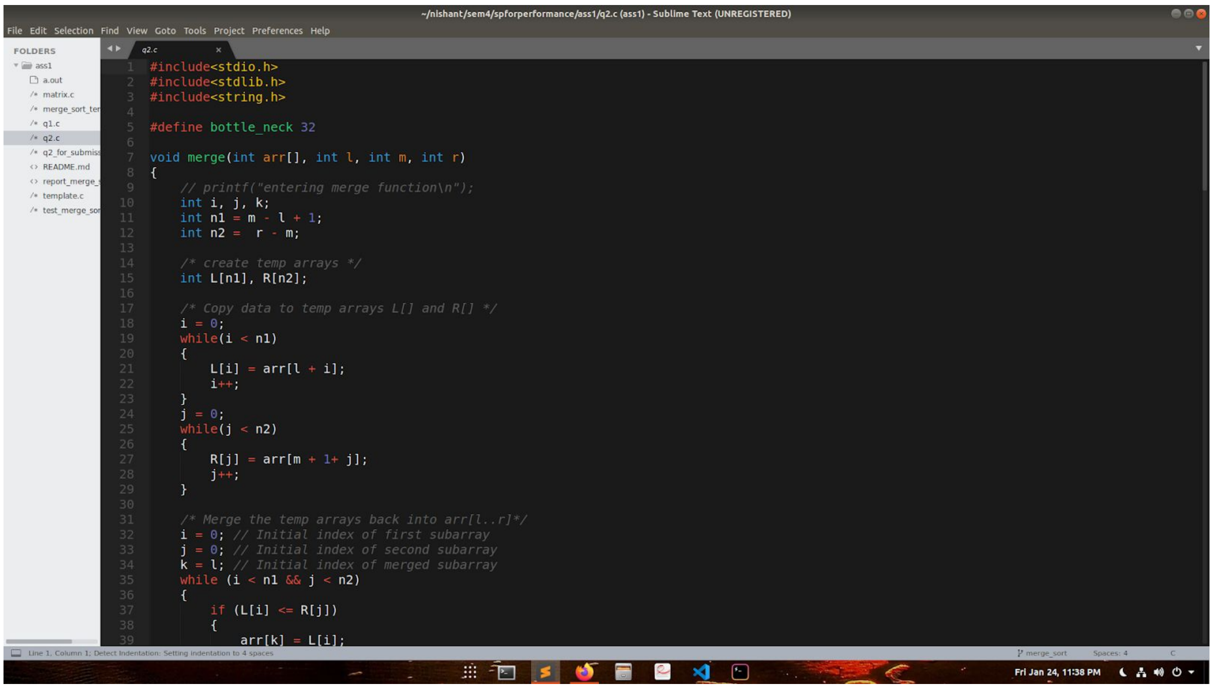
**PS** :

1. Note that the shown time take by the code varies with the fact as to whether the PC is plugged in or not.
2. This change is attributed to the fact that there are different power modes for a processor and
3. there are (privileged) ways of making it work at different performance levels so as to optimise the life of the pc wrt the power available.

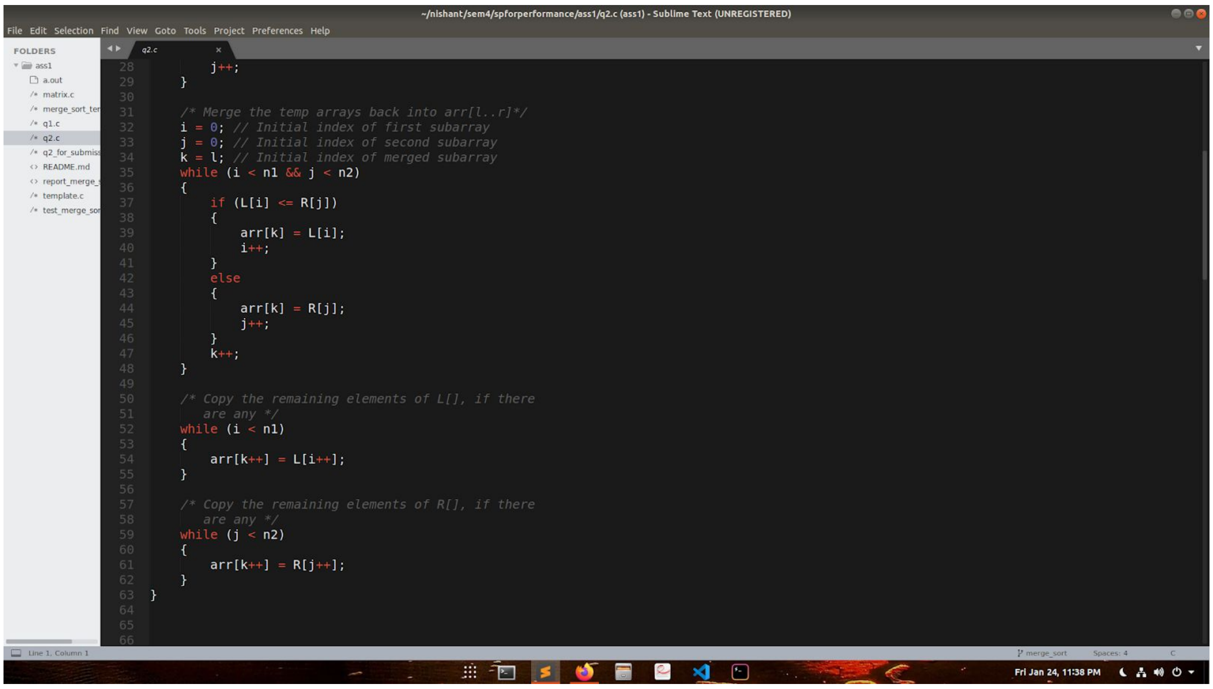
**Final Code Recorded times** :

1. Took 0.068721 seconds to execute
2. Took 0.072496 seconds to execute
3. Took 0.071928 seconds to execute

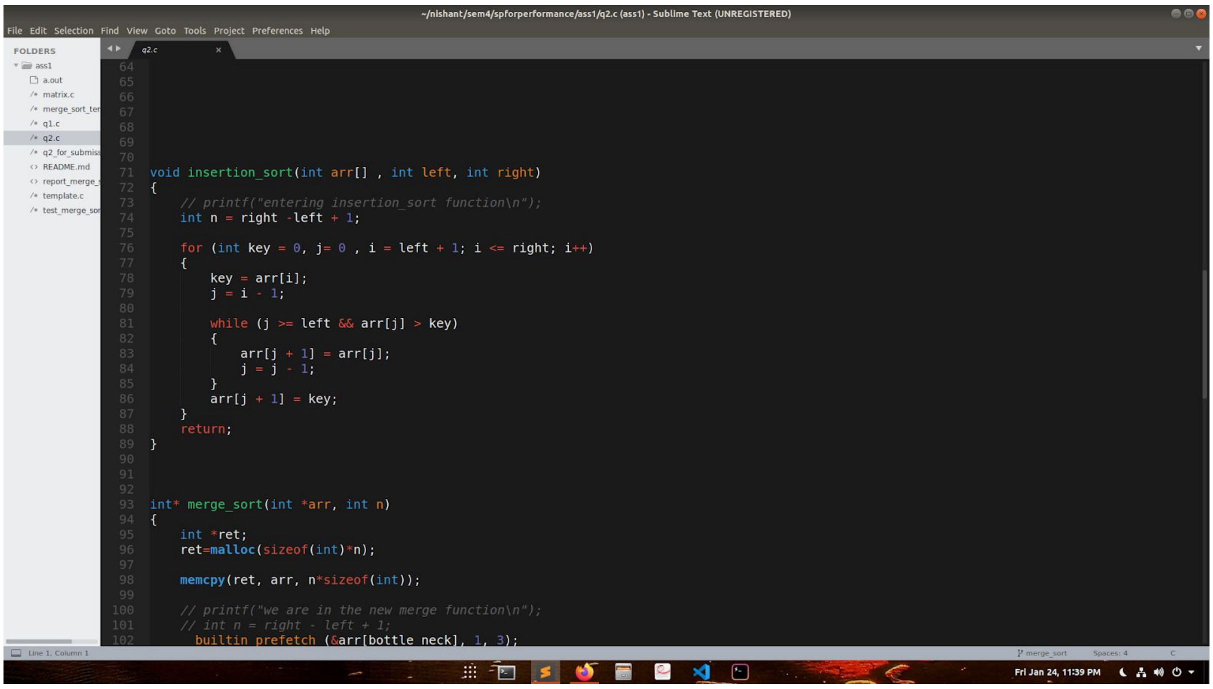
**CODE is as follows:**


~/nishant/sem4/spforperformance/ass1/q2.c (ass1) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
ass1
a.out
matrix.c
merge_sort_ter
q1.c
q2.c
q2_for_submiss
README.md
report_merge_s
template.c
test_merge_sor
Line 1, Column 1; Detect Indentation: Setting indentation to 4 spaces
merge_sort
Spaces: 4
C
Fri Jan 24, 11:38 PM

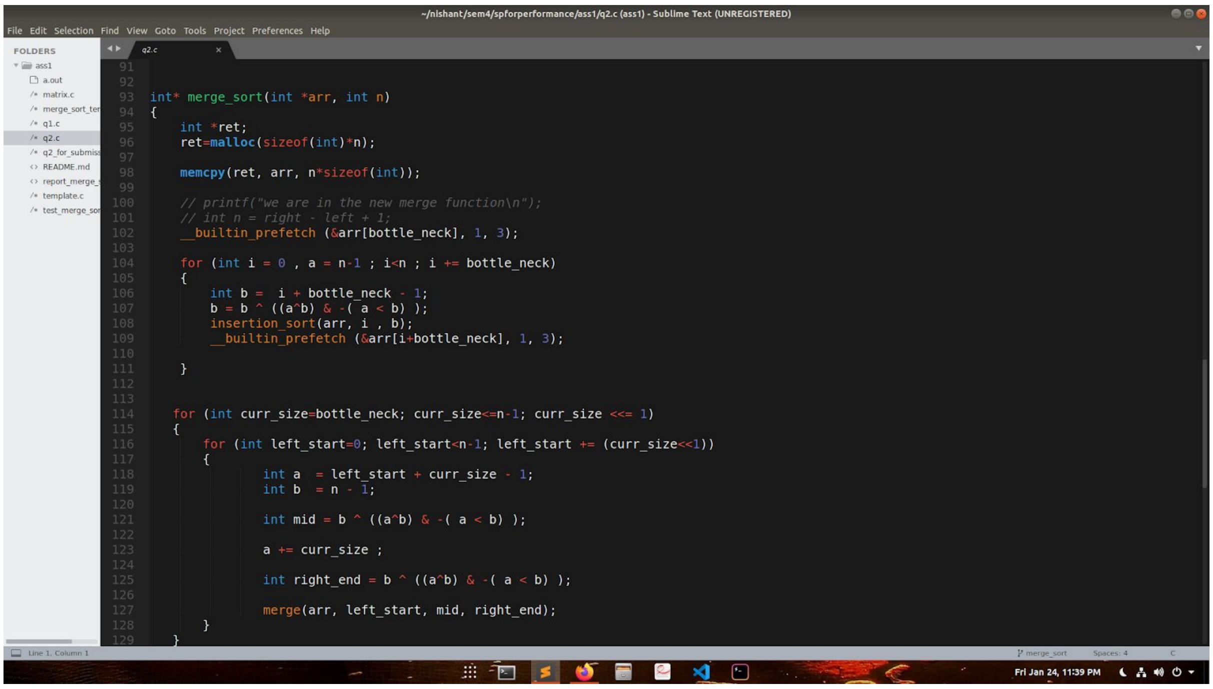
```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define bottle_neck 32

void merge(int arr[], int l, int m, int r)
{
    // printf("entering merge function\n");
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    i = 0;
    while(i < n1)
    {
        L[i] = arr[l + i];
        i++;
    }
    j = 0;
    while(j < n2)
    {
        R[j] = arr[m + 1+ j];
        j++;
    }

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
```



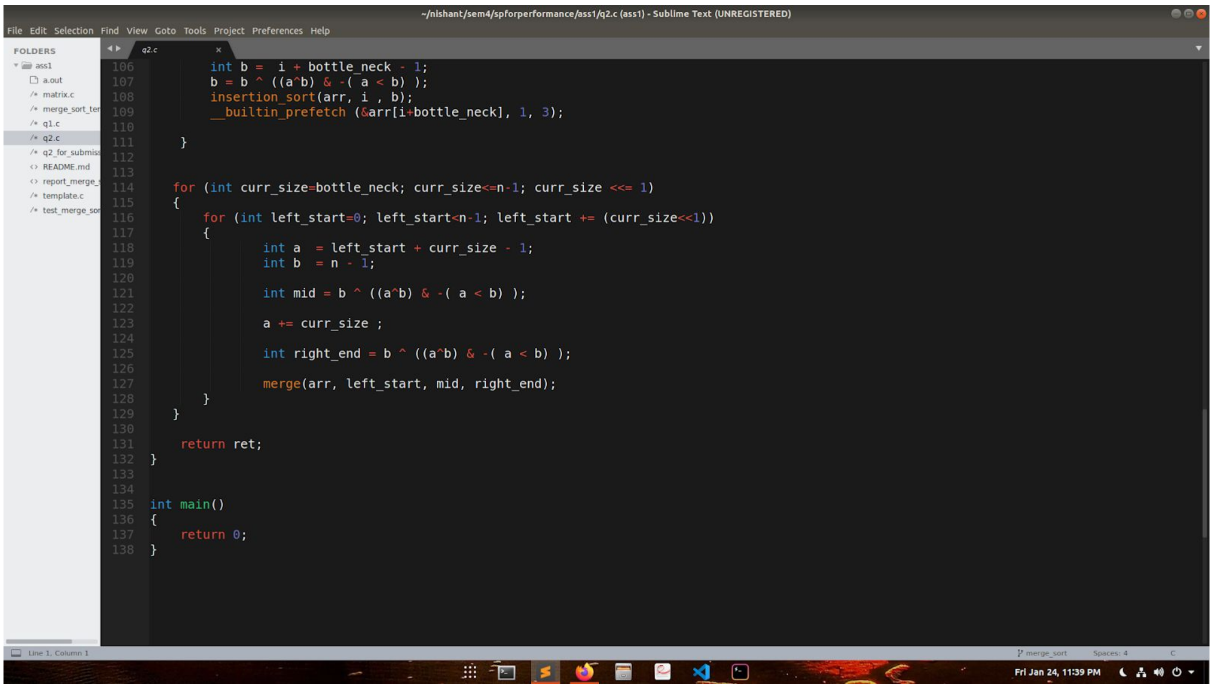
```c
        j++;
    }

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1)
    {
        arr[k++] = L[i++];
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2)
    {
        arr[k++] = R[j++];
    }
}
```



```c



void insertion_sort(int arr[] , int left, int right)
{
    // printf("entering insertion_sort function\n");
    int n = right -left + 1;

    for (int key = 0, j= 0 , i = left + 1; i <= right; i++)
    {
        key = arr[i];
        j = i - 1;

        while (j >= left && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
    return;
}


int* merge_sort(int *arr, int n)
{
    int *ret;
    ret=malloc(sizeof(int)*n);

    memcpy(ret, arr, n*sizeof(int));

    // printf("we are in the new merge function\n");
    // int n = right - left + 1;
     builtin prefetch (&arr[bottle neck], 1, 3);
```



```c


int* merge_sort(int *arr, int n)
{
    int *ret;
    ret=malloc(sizeof(int)*n);

    memcpy(ret, arr, n*sizeof(int));

    // printf("we are in the new merge function\n");
    // int n = right - left + 1;
    __builtin_prefetch (&arr[bottle_neck], 1, 3);

    for (int i = 0 , a = n-1 ; i<n ; i += bottle_neck)
    {
        int b =  i + bottle_neck - 1;
        b = b ^ ((a^b) & -( a < b) );
        insertion_sort(arr, i , b);
        __builtin_prefetch (&arr[i+bottle_neck], 1, 3);

    }


   for (int curr_size=bottle_neck; curr_size<=n-1; curr_size <<= 1)
   {
       for (int left_start=0; left_start<n-1; left_start += (curr_size<<1))
       {
               int a  = left_start + curr_size - 1;
               int b  = n - 1;

               int mid = b ^ ((a^b) & -( a < b) );

               a += curr_size ;

               int right_end = b ^ ((a^b) & -( a < b) );

               merge(arr, left_start, mid, right_end);
       }
   }
```



```c
        int b =  i + bottle_neck - 1;
        b = b ^ ((a^b) & -( a < b) );
        insertion_sort(arr, i , b);
        __builtin_prefetch (&arr[i+bottle_neck], 1, 3);

    }


   for (int curr_size=bottle_neck; curr_size<=n-1; curr_size <<= 1)
   {
       for (int left_start=0; left_start<n-1; left_start += (curr_size<<1))
       {
               int a  = left_start + curr_size - 1;
               int b  = n - 1;

               int mid = b ^ ((a^b) & -( a < b) );

               a += curr_size ;

               int right_end = b ^ ((a^b) & -( a < b) );

               merge(arr, left_start, mid, right_end);
       }
   }

    return ret;
}


int main()
{
    return 0;
}
```

## ANOTHER POSSIBLE OPTIMISATION :

## Vectorisation was a possible optimisation. :

1. Here , we use SIMD (single instruction , multiple data ) instructions to parallely process various calculations to make a much more improved version of the code.
2. Unfortunately , that hasn't been done in this code as of yet.