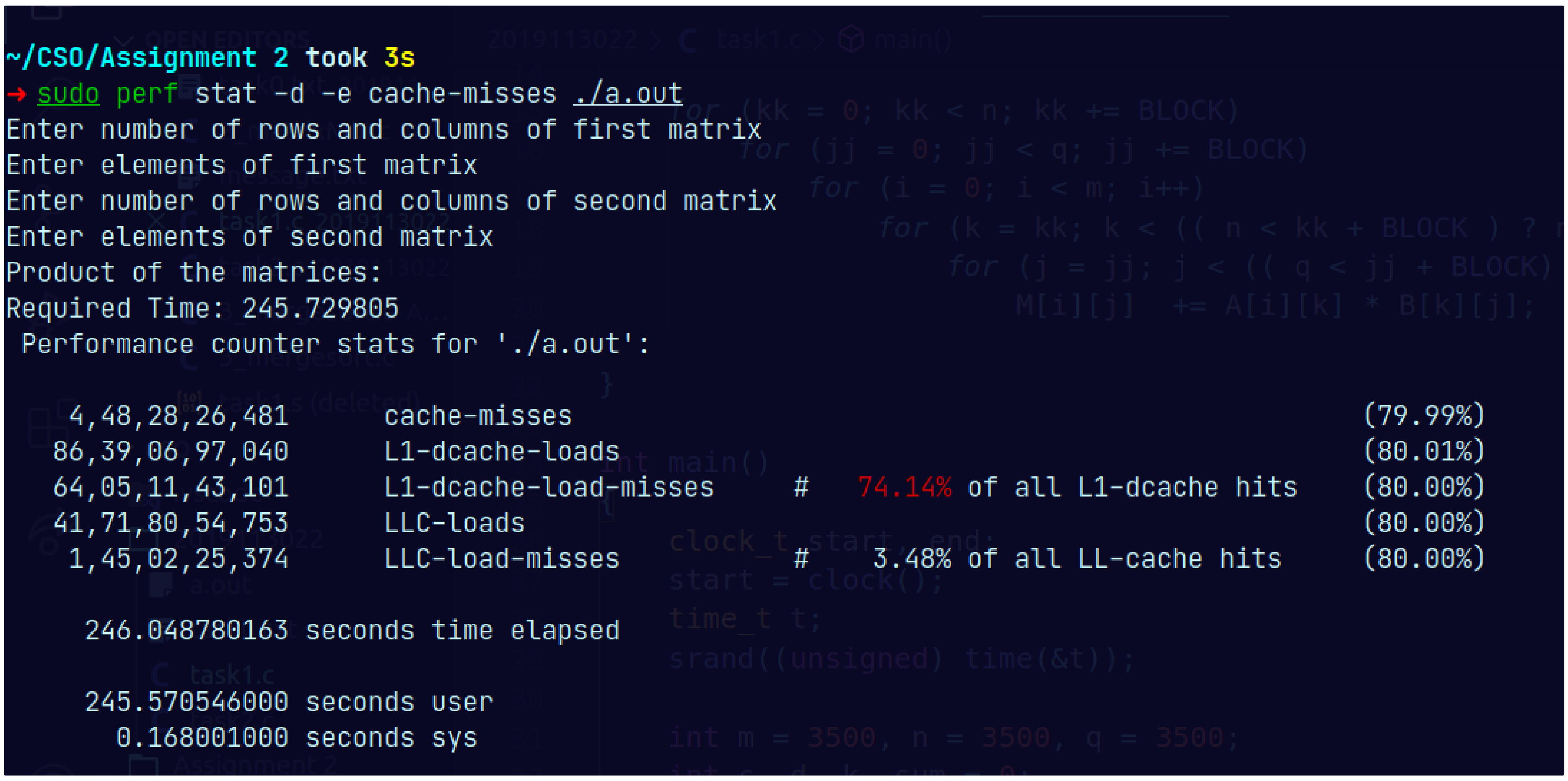
CSO Assignment - 2

Task 0 : System Information

The complete information regarding the system can be found in `task0.txt` within the directory.

Task 1: Optimising Matrix Mutiplication

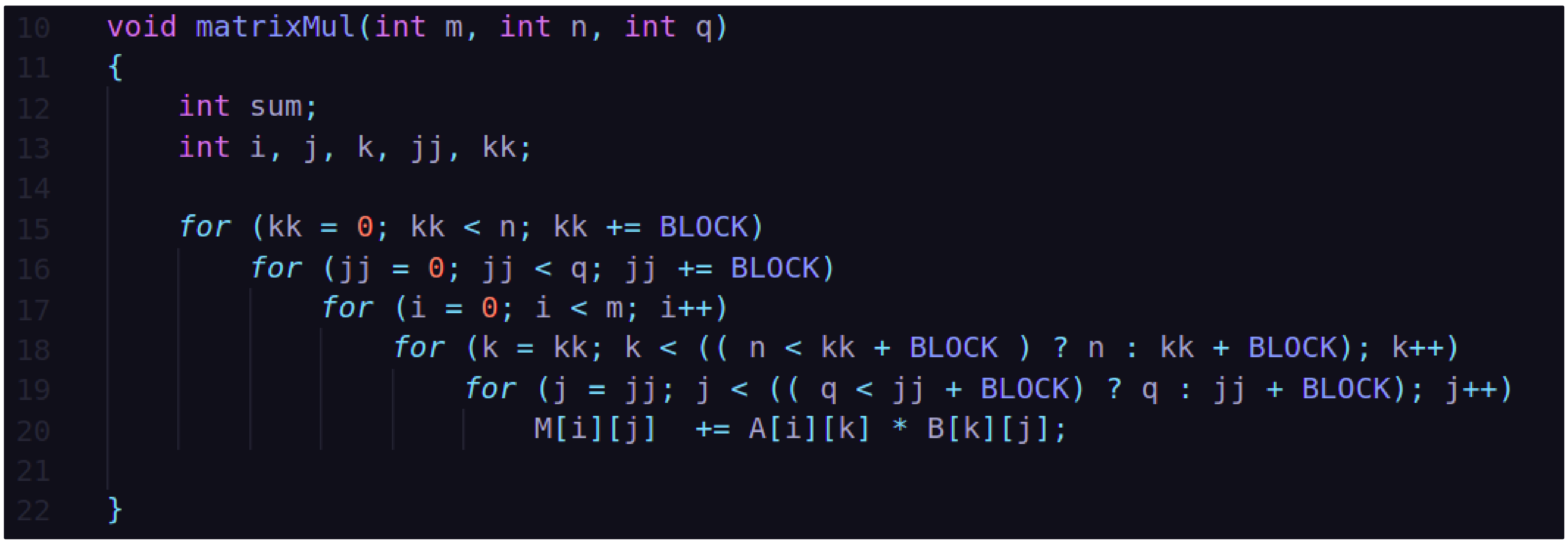
The original algorithm takes **246.048780163 seconds** with `-O2` optimisation flag and the `cache-misses` for this algorithm is **4,48,28,26,481 misses**, as given below.



```
~/CSO/Assignment 2 took 3s
➜ sudo perf stat -d -e cache-misses ./a.out
Enter number of rows and columns of first matrix
Enter elements of first matrix
Enter number of rows and columns of second matrix
Enter elements of second matrix
Product of the matrices:
Required Time: 245.729805
 Performance counter stats for './a.out':

   4,48,28,26,481      cache-misses                                                  (79.99%)
  86,39,06,97,040      L1-dcache-loads                                               (80.01%)
  64,05,11,43,101      L1-dcache-load-misses     #   74.14% of all L1-dcache hits    (80.00%)
  41,71,80,54,753      LLC-loads                                                     (80.00%)
   1,45,02,25,374      LLC-load-misses           #    3.48% of all LL-cache hits     (80.00%)

    246.048780163 seconds time elapsed

    245.570546000 seconds user
      0.168001000 seconds sys
```

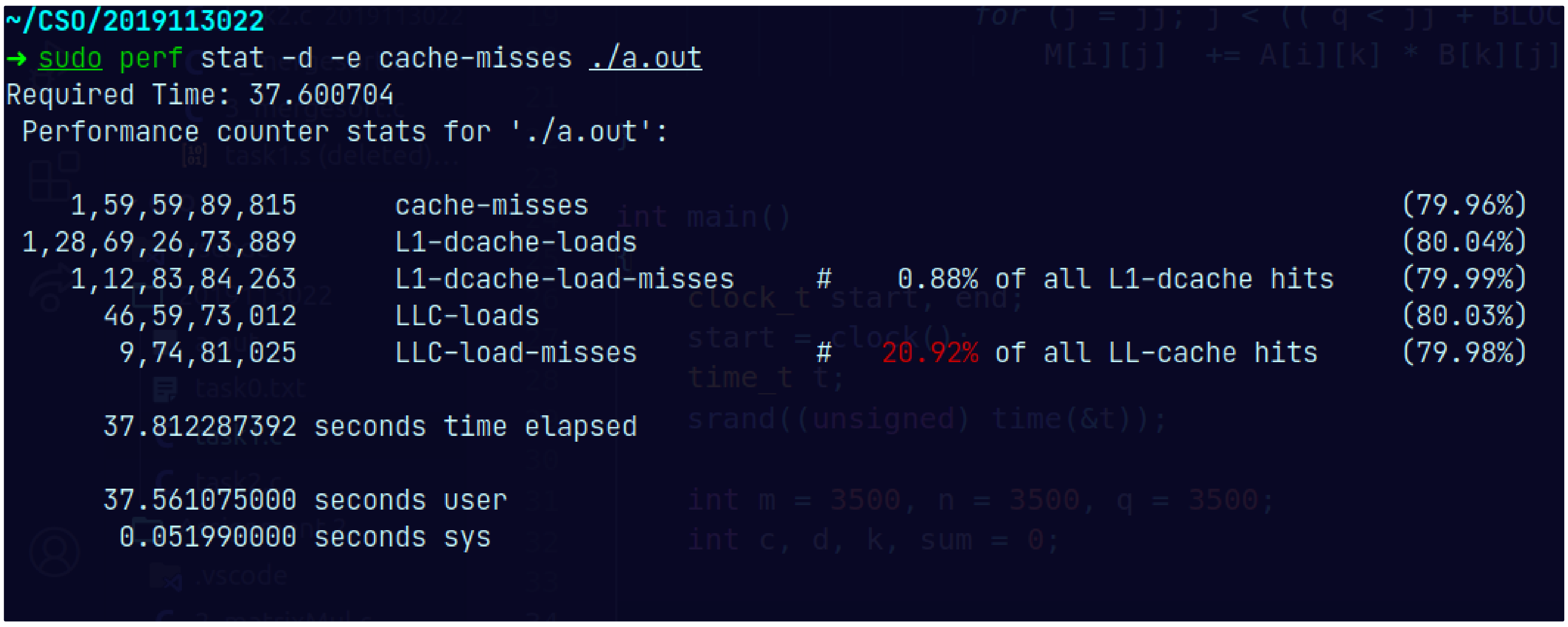
The algorithm given can be optimised in several ways to reduce the amount of cache misses and increase performance. For instance the order of the access needs to be proper, ie: for matrix multiplication for matrix $A[\,i\,][\,k\,]$ and $B[\,k\,][\,j\,]$ the order of loop should be $i,\ j$ and k.

Another possible optimisation is a simple tiling algorithm which involves dividing the matrix multiplication to smaller blocks. This means that the block in consideration will not exceed the cache size and hence will improve the performance by a great margin. The block-size I have taken here is `16`.



```
void matrixMul(int m, int n, int q)
{
    int sum;
    int i, j, k, jj, kk;

    for (kk = 0; kk < n; kk += BLOCK)
        for (jj = 0; jj < q; jj += BLOCK)
            for (i = 0; i < m; i++)
                for (k = kk; k < (( n < kk + BLOCK ) ? n : kk + BLOCK); k++)
                    for (j = jj; j < (( q < jj + BLOCK) ? q : jj + BLOCK); j++)
                        M[i][j]  += A[i][k] * B[k][j];

}
```

The modified alogrithm takes **37.812287392 seconds** with `-O2` optimisation flag and the `cache-misses` for it is **1,59,59,89,815 misses** as given below.

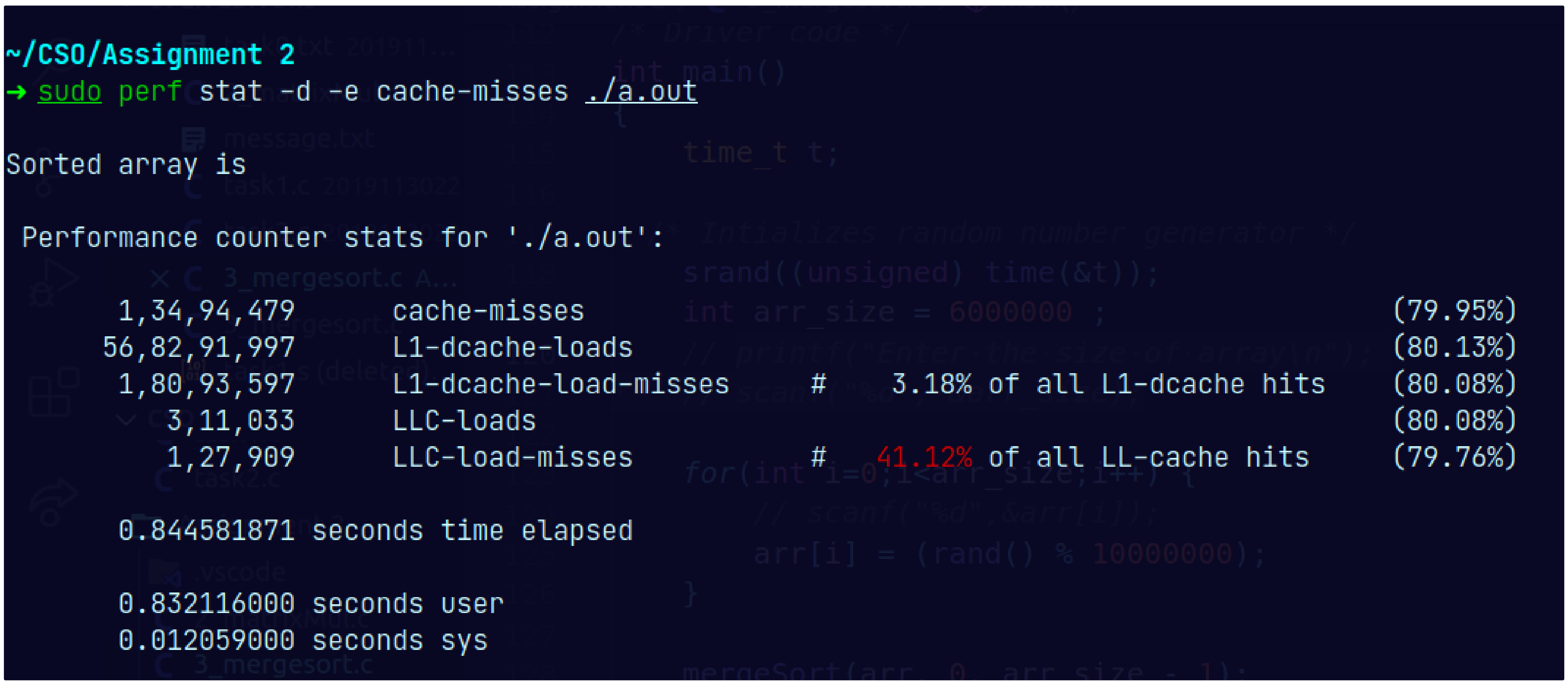


```
~/CSO/2019113022
➜ sudo perf stat -d -e cache-misses ./a.out
Required Time: 37.600704
 Performance counter stats for './a.out':

     1,59,59,89,815      cache-misses                                                   (79.96%)
  1,28,69,26,73,889      L1-dcache-loads                                                (80.04%)
     1,12,83,84,263      L1-dcache-load-misses     #    0.88% of all L1-dcache hits     (79.99%)
       46,59,73,012      LLC-loads                                                      (80.03%)
        9,74,81,025      LLC-load-misses           #   20.92% of all LL-cache hits      (79.98%)

       37.812287392 seconds time elapsed

       37.561075000 seconds user
        0.051990000 seconds sys
```

Task 2: Optimising Merge Sort

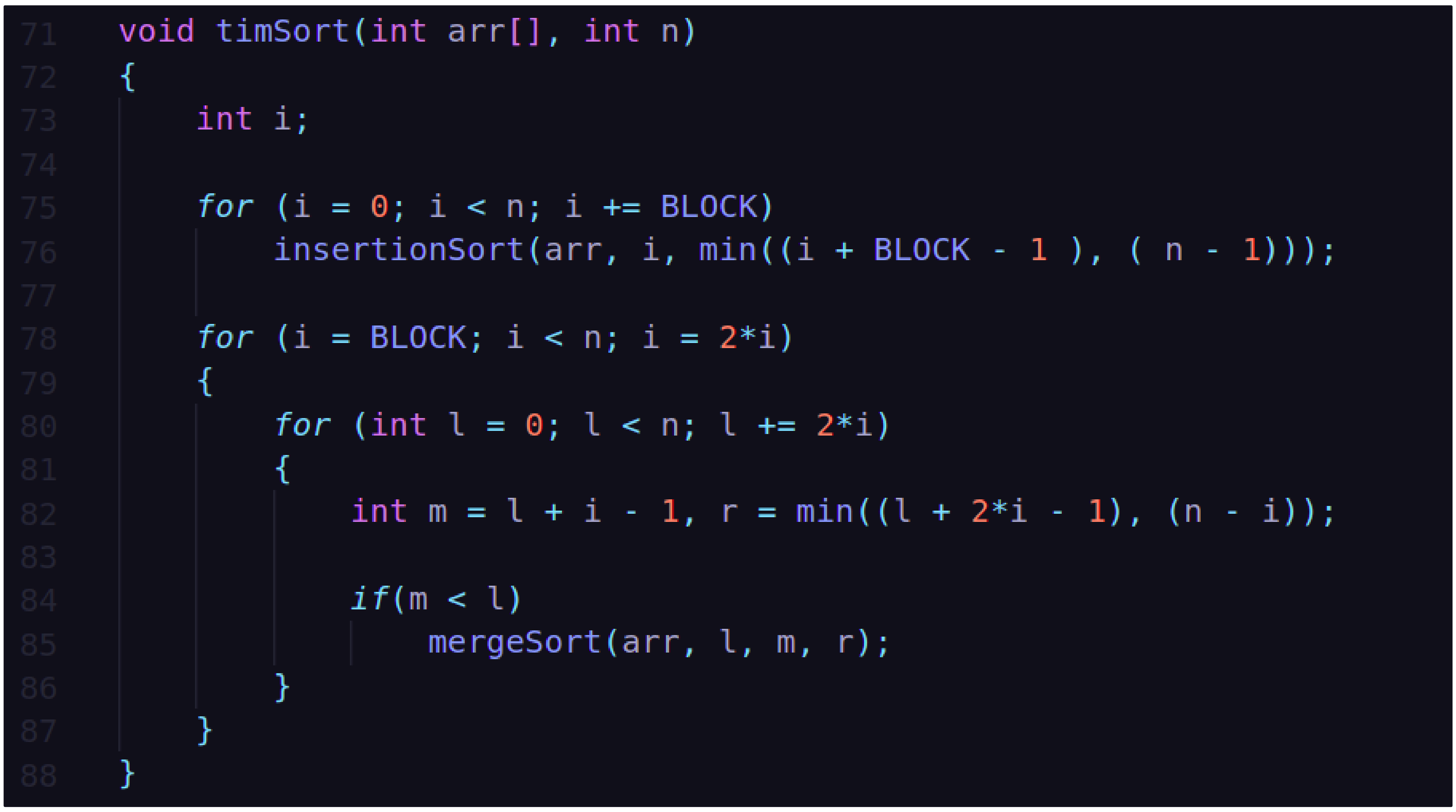
The algorithm given is originally is Merge Sort, which is a very optimal sorting algorithm, with the best possible sorting complexity of $O(n \log n)$. However, this algorithm is not very good for very small number of elements.

The original algorithm takes **0.844581871 seconds** with `-O2` optimisation flag and the `cache-misses` for this algorithm is **1,34,94,479 misses**, as given below.



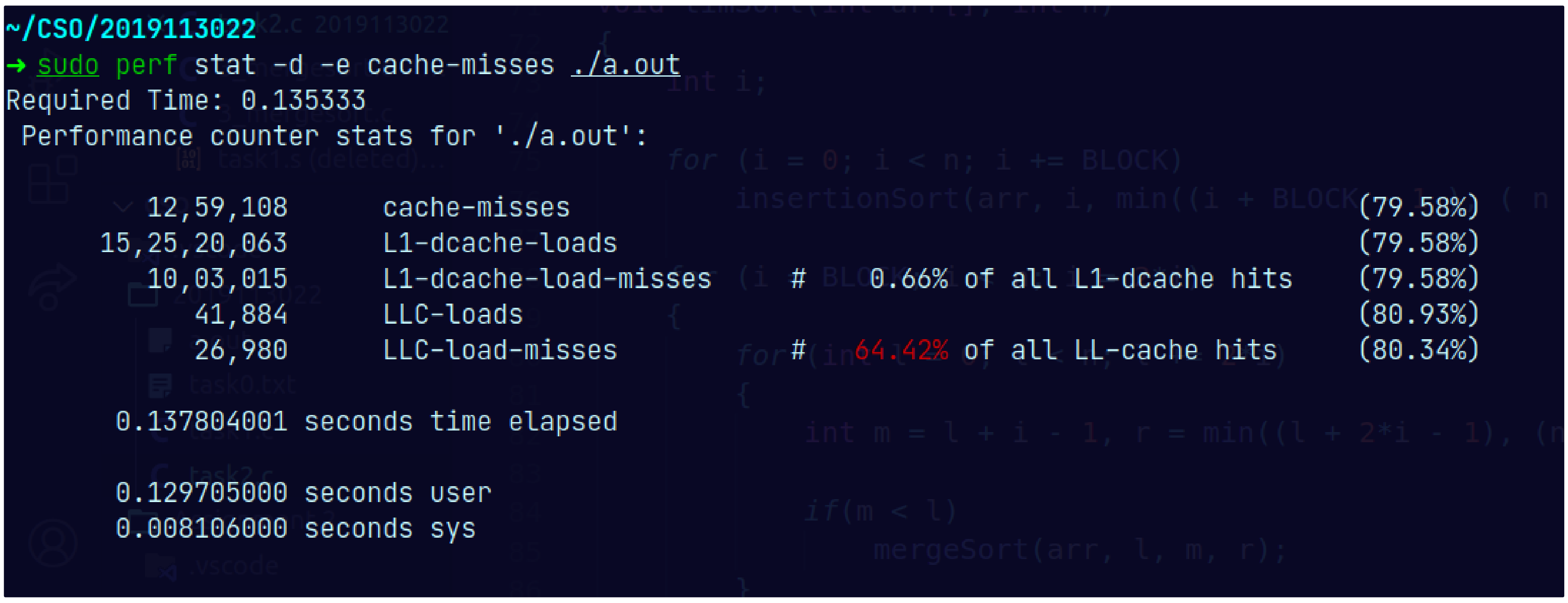
```
~/CSO/Assignment 2
➜ sudo perf stat -d -e cache-misses ./a.out

Sorted array is

 Performance counter stats for './a.out':

        1,34,94,479      cache-misses                                                  (79.95%)
       56,82,91,997      L1-dcache-loads                                               (80.13%)
        1,80,93,597      L1-dcache-load-misses     #    3.18% of all L1-dcache hits    (80.08%)
           3,11,033      LLC-loads                                                     (80.08%)
           1,27,909      LLC-load-misses           #   41.12% of all LL-cache hits     (79.76%)

        0.844581871 seconds time elapsed

        0.832116000 seconds user
        0.012059000 seconds sys
```

The work around is a sort called Tim Sort. This is a form of tiling sort, in which the array is divided into blocks, and sorted according with insertions sort. Then these arrays are then merged, similar to merge sort.



```c
void timSort(int arr[], int n)
{
    int i;

    for (i = 0; i < n; i += BLOCK)
        insertionSort(arr, i, min((i + BLOCK - 1 ), ( n - 1)));

    for (i = BLOCK; i < n; i = 2*i)
    {
        for (int l = 0; l < n; l += 2*i)
        {
            int m = l + i - 1, r = min((l + 2*i - 1), (n - i));

            if(m < l)
                mergeSort(arr, l, m, r);
        }
    }
}
```

The modified alogrithm takes **0.137804001 seconds** with `-O2` optimisation flag and the `cache-misses` for it is **12,59,108 misses** as given below.



```
~/CSO/2019113022
➜ sudo perf stat -d -e cache-misses ./a.out
Required Time: 0.135333
 Performance counter stats for './a.out':

         12,59,108      cache-misses                                                  (79.58%)
      15,25,20,063      L1-dcache-loads                                               (79.58%)
         10,03,015      L1-dcache-load-misses     #    0.66% of all L1-dcache hits    (79.58%)
            41,884      LLC-loads                                                     (80.93%)
            26,980      LLC-load-misses           #   64.42% of all LL-cache hits     (80.34%)

       0.137804001 seconds time elapsed

       0.129705000 seconds user
       0.008106000 seconds sys
```