Andrew Ma

Lab 9

CPE 435

3/15/21

Observations and Answers

Part 1.a Cachegrind

Most of the modern processors have three levels of cache. As presented above,cachegrind simulates two levels of caches. How does cachegrind handle the machines that have more than two levels of caches? Why does cachegrind do what it does for handling three levels of cache hierarchy?

For machines with more than 2 levels of caches, Cachegrind simulates the first-level (L1) and last-level (LL) caches. This is because the last-level cache has the most influence on runtime since it masks main memory accesses. The L1 caches have low associativity so we can detect cases where the code interacts badly with this cache. For 3 levels of cache hierarchy, it chooses the first and last levels.

Assignment 1

Test 1

```
x@x:~/Desktop/Lab_09$ g++ test1.cpp -o test1 && ./test1
array[0][0] was 0
```

```
x@x:~/Desktop/Lab_09$ g++ test1.cpp -o test1 && ./test1
array[0][0] was 0
```

```
x@x:~/Desktop/Lab_09$ valgrind --tool=cachegrind ./test1
==40566== Cachegrind, a cache and branch-prediction profiler
==40566== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==40566== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==40566== Command: ./test1
==40566==
--40566-- warning: L3 cache found, using its data for the LL simulation.
array[0][0] was 0
==40566==
==40566== I   refs:      13,313,301
==40566== I1  misses:         1,867
==40566== LLi misses:         1,770
==40566== I1  miss rate:       0.01%
==40566== LLi miss rate:       0.01%
==40566==
==40566== D   refs:       5,741,944  (4,548,806 rd   + 1,193,138 wr)
==40566== D1  misses:        80,398  (   15,396 rd   +    65,002 wr)
==40566== LLd misses:        72,266  (    9,232 rd   +    63,034 wr)
==40566== D1  miss rate:        1.4% (      0.3%     +       5.4%  )
==40566== LLd miss rate:        1.3% (      0.2%     +       5.3%  )
==40566==
==40566== LL refs:           82,265  (   17,263 rd   +    65,002 wr)
==40566== LL misses:         74,036  (   11,002 rd   +    63,034 wr)
==40566== LL miss rate:         0.4% (      0.1%     +       5.3%  )
x@x:~/Desktop/Lab_09$
```

Test 2

```
x@x:~/Desktop/Lab_09$ valgrind --tool=cachegrind ./test2
==40574== Cachegrind, a cache and branch-prediction profiler
==40574== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==40574== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==40574== Command: ./test2
==40574==
--40574-- warning: L3 cache found, using its data for the LL simulation.
array[0][0] was 0
==40574==
==40574== I   refs:      13,313,301
==40574== I1  misses:         1,867
==40574== LLi misses:         1,770
==40574== I1  miss rate:       0.01%
==40574== LLi miss rate:       0.01%
==40574==
==40574== D   refs:       5,741,944  (4,548,806 rd   + 1,193,138 wr)
==40574== D1  misses:     1,017,932  (   15,396 rd   + 1,002,536 wr)
==40574== LLd misses:        72,266  (    9,232 rd   +    63,034 wr)
==40574== D1  miss rate:       17.7% (      0.3%     +      84.0%  )
==40574== LLd miss rate:        1.3% (      0.2%     +       5.3%  )
==40574==
==40574== LL refs:        1,019,799  (   17,263 rd   + 1,002,536 wr)
==40574== LL misses:         74,036  (   11,002 rd   +    63,034 wr)
==40574== LL miss rate:         0.4% (      0.1%     +       5.3%  )
x@x:~/Desktop/Lab_09$
```

The differences start in D1 misses. Test 1 had 80,398 D1 misses, and Test 2 had 1,017,932 D1 misses. Test 1 had 1.4% D1 miss rate while Test 2 had 17.7% D1 miss rate. There was also a difference in LL refs. Test 1 had 82,265 LL refs, while Test 2 had 1,019,799 LL refs.

1. Test 1 is better in terms of performance because it has fewer D1 misses.
2. Because it has fewer D1 misses, this means there are less accesses to memory and more cache hits, so it is faster.

Part 1.b memcheck

Assignment 3

```
x@x:~/Desktop/Lab_09$ gcc test3.c -o test3 && ./test3
x@x:~/Desktop/Lab_09$
```

```
x@x:~/Desktop/Lab_09$ gcc test4.c -o test4 && ./test4
x@x:~/Desktop/Lab_09$
```

```
x@x:~/Desktop/Lab_09$ gcc test5.c -o test5 && ./test5
X is zero
```

Assignment 4

```
x@x:~/Desktop/Lab_09$ valgrind --tool=memcheck --leak-check=yes ./test3
==41130== Memcheck, a memory error detector
==41130== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==41130== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==41130== Command: ./test3
==41130==
==41130==
==41130== HEAP SUMMARY:
==41130==     in use at exit: 100 bytes in 1 blocks
==41130==   total heap usage: 1 allocs, 0 frees, 100 bytes allocated
==41130==
==41130== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==41130==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==41130==    by 0x10915E: main (in /home/x/Desktop/Lab_09/test3)
==41130==
==41130== LEAK SUMMARY:
==41130==    definitely lost: 100 bytes in 1 blocks
==41130==    indirectly lost: 0 bytes in 0 blocks
==41130==      possibly lost: 0 bytes in 0 blocks
==41130==    still reachable: 0 bytes in 0 blocks
==41130==         suppressed: 0 bytes in 0 blocks
==41130==
==41130== For lists of detected and suppressed errors, rerun with: -s
==41130== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
x@x:~/Desktop/Lab_09$
```

For test3 the error is memory leak, since we ran malloc and allocated 100 bytes, but we never ran free(x).

```
x@x:~/Desktop/Lab_09$ valgrind --tool=memcheck --leak-check=yes ./test4
==41136== Memcheck, a memory error detector
==41136== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==41136== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==41136== Command: ./test4
==41136==
==41136== Invalid write of size 1
==41136==    at 0x10916B: main (in /home/x/Desktop/Lab_09/test4)
==41136==  Address 0x4a4f04a is 0 bytes after a block of size 10 alloc'd
==41136==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==41136==    by 0x10915E: main (in /home/x/Desktop/Lab_09/test4)
==41136==
==41136==
==41136== HEAP SUMMARY:
==41136==     in use at exit: 10 bytes in 1 blocks
==41136==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==41136==
==41136== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1
==41136==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==41136==    by 0x10915E: main (in /home/x/Desktop/Lab_09/test4)
==41136==
==41136== LEAK SUMMARY:
==41136==    definitely lost: 10 bytes in 1 blocks
==41136==    indirectly lost: 0 bytes in 0 blocks
==41136==      possibly lost: 0 bytes in 0 blocks
==41136==    still reachable: 0 bytes in 0 blocks
==41136==         suppressed: 0 bytes in 0 blocks
==41136==
==41136== For lists of detected and suppressed errors, rerun with: -s
==41136== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
x@x:~/Desktop/Lab_09$
```

For test4 the error is also memory leak, since we ran malloc and allocated 10 bytes, but we never ran free(x). Another error is an invalid write since we tried to write ‘a’ to index 10, but the last valid index was 9.

```
x@x:~/Desktop/Lab_09$ valgrind --tool=memcheck --leak-check=yes ./test5
==41144== Memcheck, a memory error detector
==41144== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==41144== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==41144== Command: ./test5
==41144==
==41144== Conditional jump or move depends on uninitialised value(s)
==41144==    at 0x109159: main (in /home/x/Desktop/Lab_09/test5)
==41144==
X is zero
==41144==
==41144== HEAP SUMMARY:
==41144==     in use at exit: 0 bytes in 0 blocks
==41144==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==41144==
==41144== All heap blocks were freed -- no leaks are possible
==41144==
==41144== Use --track-origins=yes to see where uninitialised values come from
==41144== For lists of detected and suppressed errors, rerun with: -s
==41144== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
x@x:~/Desktop/Lab_09$
```

For test5 the error is uninitialized value, since we never initialized x but used it for comparing to 0.

Part 2 Power Consumption measurement Using Hardware Monitor Pro

Assignment 5

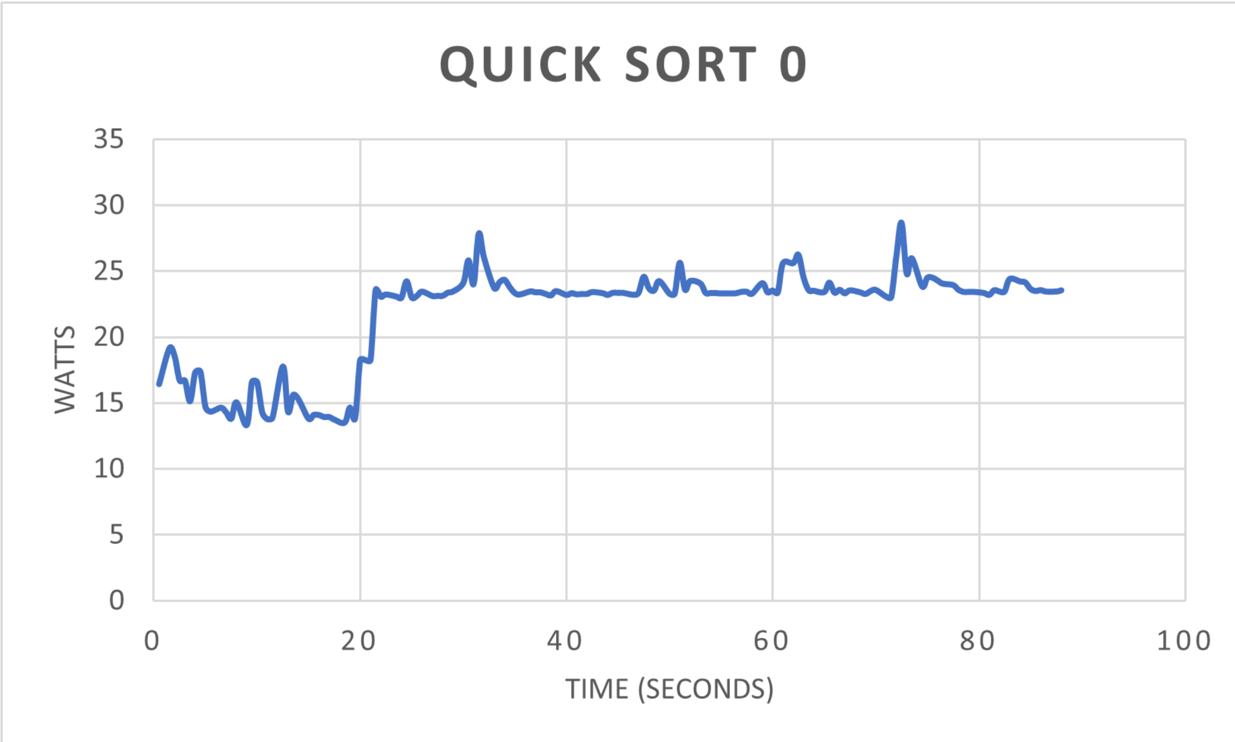
Look in source code below for quick sort, merge sort, and insertion sort code

```
x ~ SHARES UBUNTU_SHARED Lab_09 1 make
g++ -O0 quicksort.cpp -o quicksort_0
g++ -O0 mergesort.cpp -o mergesort_0
g++ -O0 insertionsort.cpp -o insertionsort_0
g++ -O3 quicksort.cpp -o quicksort_3
g++ -O3 mergesort.cpp -o mergesort_3
g++ -O3 insertionsort.cpp -o insertionsort_3
```

Assignment 6

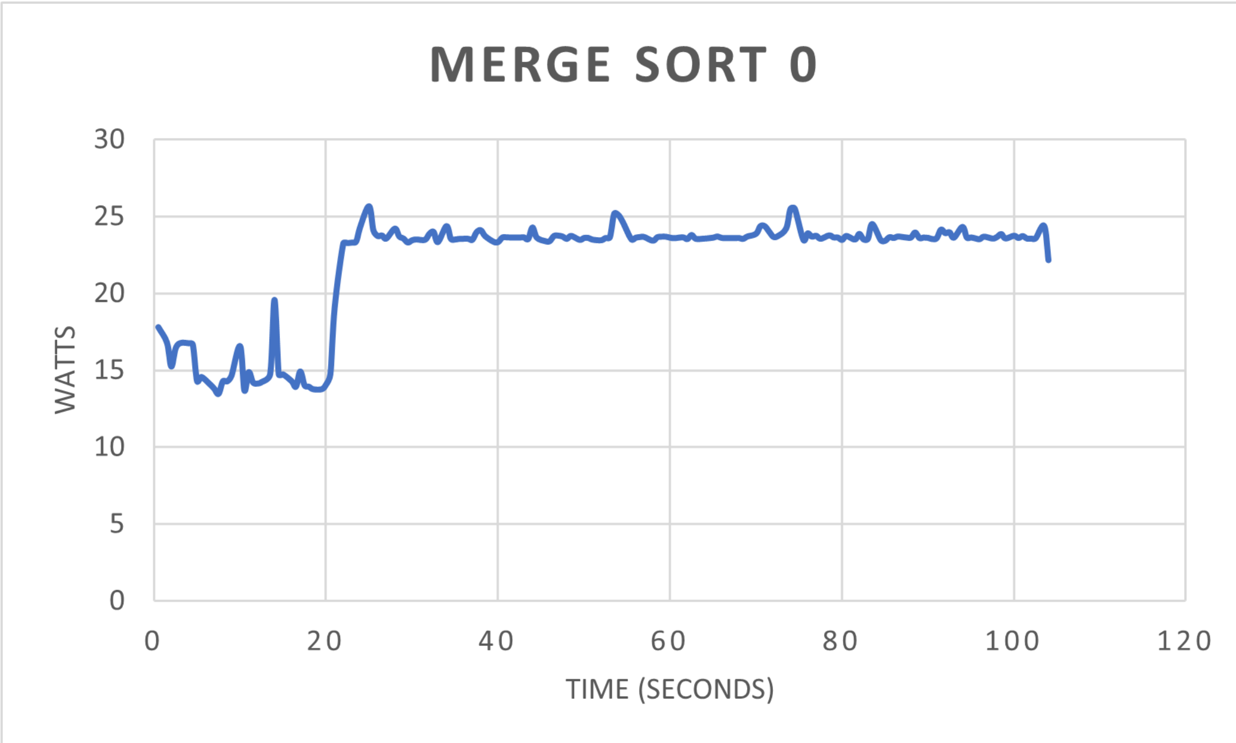
Quick Sort Optimization 0

```
x ~ SHARES UBUNTU_SHARED Lab_09 ./quicksort_0
Quick Sort [1]  sorted: true    Took 13630 ms   n=50000000
Quick Sort [2]  sorted: true    Took 13432 ms   n=50000000
Quick Sort [3]  sorted: true    Took 13785 ms   n=50000000
Quick Sort [4]  sorted: true    Took 13609 ms   n=50000000
Quick Sort [5]  sorted: true    Took 13597 ms   n=50000000
Average Milliseconds: 13610 ms
Average Milliseconds per element: 0.0002722 ms/element
x ~ SHARES UBUNTU_SHARED Lab_09
```



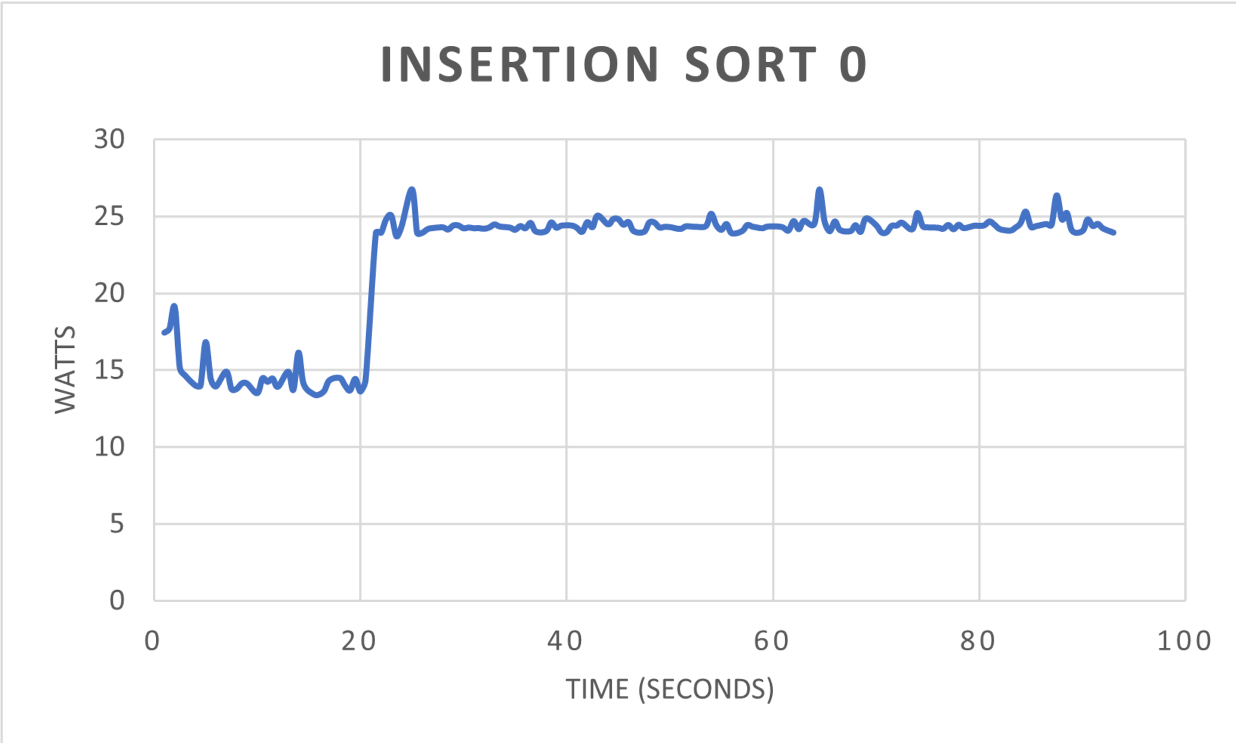
Merge Sort Optimization 0

```
x ~ 〉 SHARES 〉 UBUNTU_SHARED 〉 Lab_09 ./mergesort_0
Merge Sort [1]  sorted: true    Took 16803 ms   n=50000000
Merge Sort [2]  sorted: true    Took 16713 ms   n=50000000
Merge Sort [3]  sorted: true    Took 16715 ms   n=50000000
Merge Sort [4]  sorted: true    Took 16701 ms   n=50000000
Merge Sort [5]  sorted: true    Took 16685 ms   n=50000000
Average Milliseconds: 16723 ms
Average Milliseconds per element: 0.00033446 ms/element
x ~ 〉 SHARES 〉 UBUNTU_SHARED 〉 Lab_09
```



Insertion Sort Optimization 0

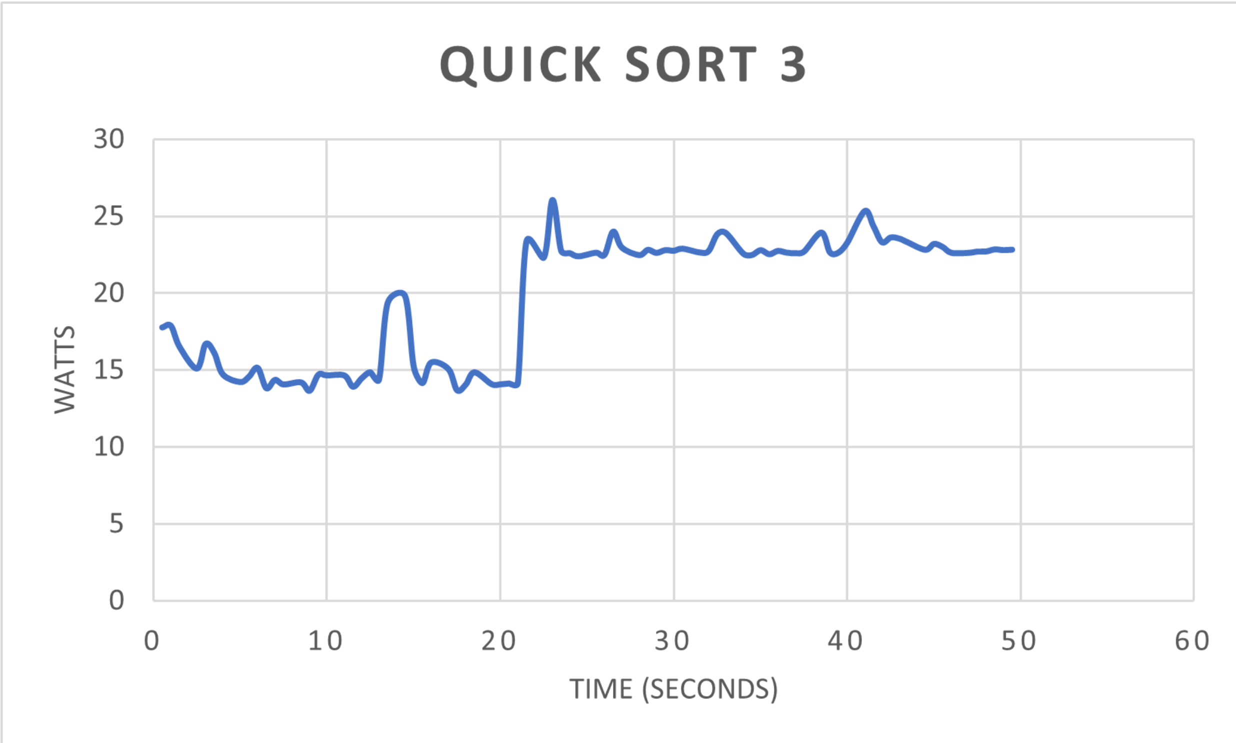
```
x  ~ > SHARES > UBUNTU_SHARED > Lab_09 > 130  ./insertionsort_0
Insertion Sort [1]      sorted: true    Took 14740 ms    n=140000
Insertion Sort [2]      sorted: true    Took 14626 ms    n=140000
Insertion Sort [3]      sorted: true    Took 14603 ms    n=140000
Insertion Sort [4]      sorted: true    Took 14515 ms    n=140000
Insertion Sort [5]      sorted: true    Took 14585 ms    n=140000
Average Milliseconds: 14613 ms
Average Milliseconds per element: 0.104379 ms/element
x  ~ > SHARES > UBUNTU_SHARED > Lab_09 >
```



Assignment 7

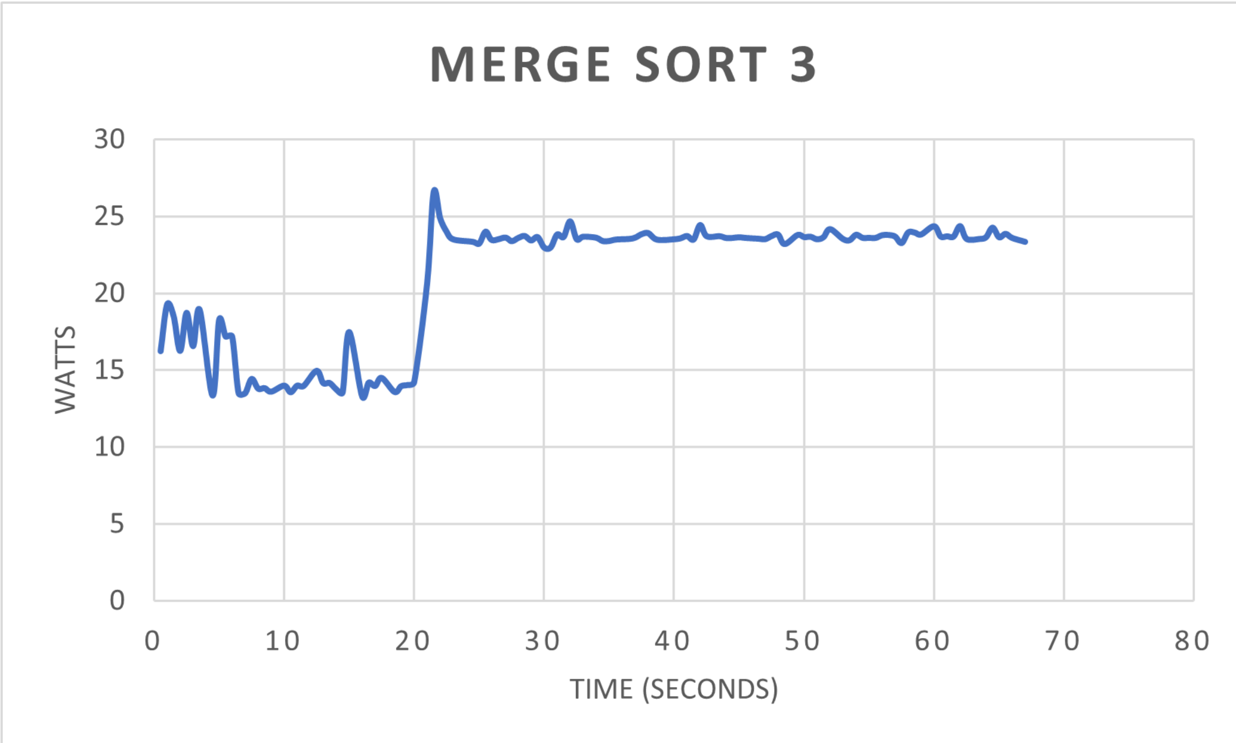
Quick Sort Optimization 3

```
x  ~ 〉 SHARES 〉 UBUNTU_SHARED 〉 Lab_09   ./quicksort_3
Quick Sort [1]  sorted: true    Took 5791 ms    n=50000000
Quick Sort [2]  sorted: true    Took 5712 ms    n=50000000
Quick Sort [3]  sorted: true    Took 5739 ms    n=50000000
Quick Sort [4]  sorted: true    Took 5861 ms    n=50000000
Quick Sort [5]  sorted: true    Took 5760 ms    n=50000000
Average Milliseconds: 5772 ms
Average Milliseconds per element: 0.00011544 ms/element
x  ~ 〉 SHARES 〉 UBUNTU_SHARED 〉 Lab_09
```

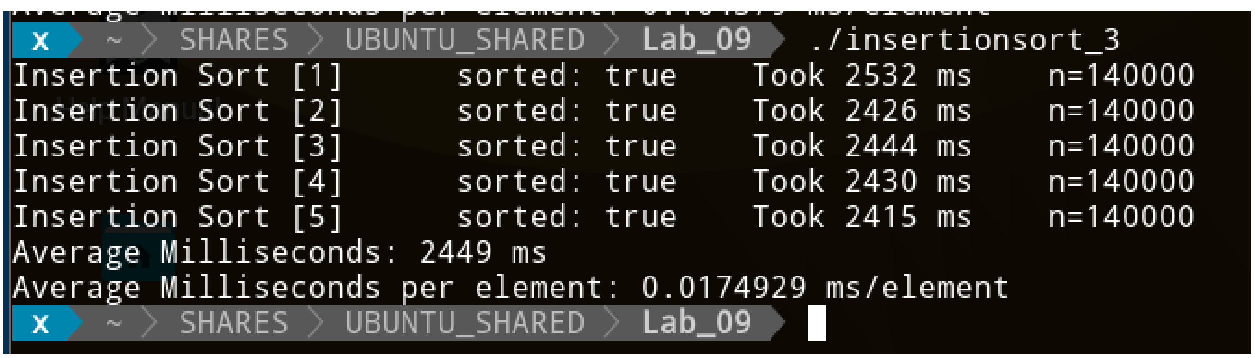


Merge Sort Optimization 3

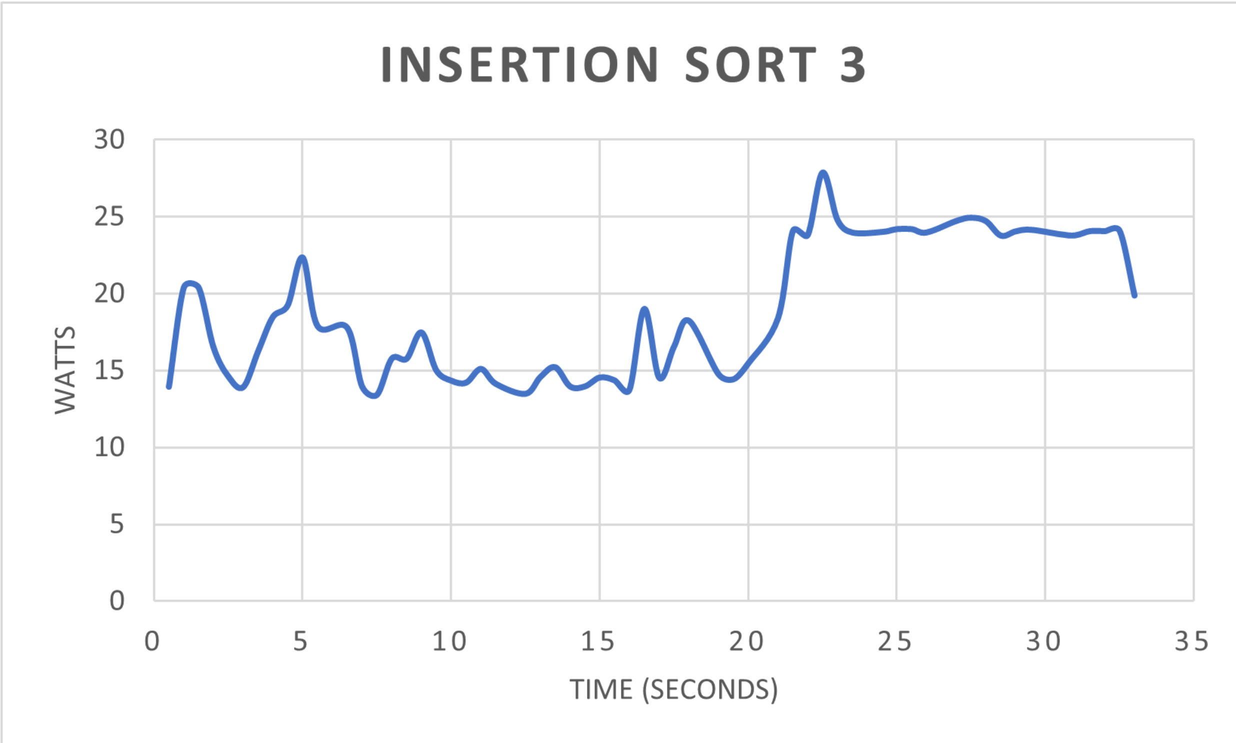
```
 x  ~ 〉 SHARES 〉 UBUNTU_SHARED 〉 Lab_09  ./mergesort_3
Merge Sort [1]  sorted: true    Took 9461 ms    n=50000000
Merge Sort [2]  sorted: true    Took 9274 ms    n=50000000
Merge Sort [3]  sorted: true    Took 9292 ms    n=50000000
Merge Sort [4]  sorted: true    Took 9308 ms    n=50000000
Merge Sort [5]  sorted: true    Took 9349 ms    n=50000000
Average Milliseconds: 9336 ms
Average Milliseconds per element: 0.00018672 ms/element
 x  ~ 〉 SHARES 〉 UBUNTU_SHARED 〉 Lab_09
```



Insertion Sort Optimization 3



```
x ~ SHARES UBUNTU_SHARED Lab_09 ./insertionsort_3
Insertion Sort [1]      sorted: true    Took 2532 ms    n=140000
Insertion Sort [2]      sorted: true    Took 2426 ms    n=140000
Insertion Sort [3]      sorted: true    Took 2444 ms    n=140000
Insertion Sort [4]      sorted: true    Took 2430 ms    n=140000
Insertion Sort [5]      sorted: true    Took 2415 ms    n=140000
Average Milliseconds: 2449 ms
Average Milliseconds per element: 0.0174929 ms/element
x ~ SHARES UBUNTU_SHARED Lab_09
```


INSERTION SORT 3
30
25
20
15
10
5
0
WATTS
0
5
10
15
20
25
30
35
TIME (SECONDS)

```
insertionsort_0
insertionsort_0 [1]:    avg_watts=23.553       elapsed_seconds=14.74   avg_joules=347.179
insertionsort_0 [2]:    avg_watts=24.392       elapsed_seconds=14.626  avg_joules=356.759
insertionsort_0 [3]:    avg_watts=24.337       elapsed_seconds=14.603  avg_joules=355.387
insertionsort_0 [4]:    avg_watts=24.448       elapsed_seconds=14.515  avg_joules=354.856
insertionsort_0 [5]:    avg_watts=24.510       elapsed_seconds=14.585  avg_joules=357.477
insertionsort_0 total:  avg_joules=354.332      avg_joules/element=0.002531

insertionsort_3
insertionsort_3 [1]:    avg_watts=21.924       elapsed_seconds=2.532   avg_joules=55.511
insertionsort_3 [2]:    avg_watts=24.264       elapsed_seconds=2.426   avg_joules=58.864
insertionsort_3 [3]:    avg_watts=24.276       elapsed_seconds=2.444   avg_joules=59.331
insertionsort_3 [4]:    avg_watts=24.329       elapsed_seconds=2.43    avg_joules=59.118
insertionsort_3 [5]:    avg_watts=23.947       elapsed_seconds=2.415   avg_joules=57.833
insertionsort_3 total:  avg_joules=58.131       avg_joules/element=0.000415

mergesort_0
mergesort_0 [1]:        avg_watts=22.893       elapsed_seconds=16.803  avg_joules=384.665
mergesort_0 [2]:        avg_watts=23.678       elapsed_seconds=16.713  avg_joules=395.730
mergesort_0 [3]:        avg_watts=23.707       elapsed_seconds=16.715  avg_joules=396.266
mergesort_0 [4]:        avg_watts=23.875       elapsed_seconds=16.701  avg_joules=398.735
mergesort_0 [5]:        avg_watts=23.716       elapsed_seconds=16.685  avg_joules=395.706
mergesort_0 total:      avg_joules=394.220      avg_joules/element=0.000008

mergesort_3
mergesort_3 [1]:        avg_watts=23.051       elapsed_seconds=9.461   avg_joules=218.082
mergesort_3 [2]:        avg_watts=23.616       elapsed_seconds=9.274   avg_joules=219.012
mergesort_3 [3]:        avg_watts=23.675       elapsed_seconds=9.292   avg_joules=219.990
mergesort_3 [4]:        avg_watts=23.664       elapsed_seconds=9.308   avg_joules=220.267
mergesort_3 [5]:        avg_watts=23.815       elapsed_seconds=9.349   avg_joules=222.647
mergesort_3 total:      avg_joules=219.999      avg_joules/element=0.000004
```

```
quicksort_0
quicksort_0 [1]:        avg_watts=23.194       elapsed_seconds=13.63   avg_joules=316.132
quicksort_0 [2]:        avg_watts=23.391       elapsed_seconds=13.432  avg_joules=314.190
quicksort_0 [3]:        avg_watts=23.695       elapsed_seconds=13.785  avg_joules=326.636
quicksort_0 [4]:        avg_watts=24.367       elapsed_seconds=13.609  avg_joules=331.608
quicksort_0 [5]:        avg_watts=23.760       elapsed_seconds=13.597  avg_joules=323.066
quicksort_0 total:      avg_joules=322.327      avg_joules/element=0.000006

quicksort_3
quicksort_3 [1]:        avg_watts=20.432       elapsed_seconds=5.791   avg_joules=118.321
quicksort_3 [2]:        avg_watts=22.835       elapsed_seconds=5.712   avg_joules=130.434
quicksort_3 [3]:        avg_watts=22.873       elapsed_seconds=5.739   avg_joules=131.270
quicksort_3 [4]:        avg_watts=23.531       elapsed_seconds=5.861   avg_joules=137.915
quicksort_3 [5]:        avg_watts=22.858       elapsed_seconds=5.76    avg_joules=131.661
quicksort_3 total:      avg_joules=129.920      avg_joules/element=0.000003
```

Round	Quick Sort 0	Merge Sort 0	Insertion Sort 0
1	**316.132**	**384.665**	**347.179**
2	**314.19**	**395.73**	**356.759**
3	**326.636**	**396.266**	**355.387**
4	**331.608**	**398.735**	**354.856**
5	**323.066**	**395.706**	**357.477**
Average Joules	**322.326**	**394.220**	**354.332**
Average Joules Per Element	0.000006	0.000008	0.002531
Round	**Quick Sort 3**	**Merge Sort 3**	**Insertion Sort 3**
1	**118.321**	**218.082**	**55.511**
2	**130.434**	**219.012**	**58.864**
3	**131.27**	**219.99**	**59.331**
4	**137.915**	**220.267**	**59.118**
5	**131.661**	**222.647**	**57.833**
Average Joules	**129.920**	**220.000**	**58.131**
Average Joules Per Element	0.000003	0.000004	0.000415

Assignment 8

The quick sort with optimization 3 was the best in terms of energy consumption per element. Yes, the compiler flag improved it from 0.000006 joules per element to 0.0000003 joules per element. The optimization compiler flag improved the energy usage for the other algorithms as well.

Extra Credit

1. The power consumption (measured in watts) is roughly the same because all the algorithms are running on the same CPU and the CPU's power usage is going to be around the same when it is running them at around the same frequency (since it maxes out the CPU and runs on max clock speed).
2. To determine if the hardware monitor is corrupting the power measurement, we can use test equipment to get the voltage the CPU is running at, and the current it draws. From this, we can calculate the power measurement with power = voltage * current. If we are just trying to see if a specific hardware monitor software is faulty, then we could compare it with another hardware monitor's power measurement or write our own program that accesses the operating system's power measurements directly.
3. MSR stands for Model Specific Registers, and these are control registers used in the x86 instruction set for debugging, program execution tracing, monitoring computer performance, and enabling or disabling certain CPU features. We can read and write to MSR registers with the rdmsr and wrmsr instructions. These can only be executed by the operating system since these are privileged. In Linux, a msr kernel module can be accessed through the pseudo file /dev/cpu/x/msr and users can read or write to this file to access these registers.

Conclusion

The lab worked as expected, and in part 1 I learned how to use valgrind with the cachegrind and memcheck tools. This makes it easy to check for causes of errors dynamically instead of analyzing the code statically. In part 2 of the lab, I learned how to use a hardware monitor software to measure the CPU package power usage. I also was able to integrate the measurements with Python code to calculate the watts and joules for each round, and then calculate the average joules per element to determine which of the algorithms was the best in terms of power usage. The quick sort was the fastest with the merge sort slightly behind it but still very fast, and the insertion sort was the slowest because it has a quadratic time complexity instead of nlogn. Also, changing the optimization compilation flag had a significant improvement in all the programs.

Source Code

```
/* test1.cpp */
#include <iostream>

using namespace std;

int main() {
    int array[1000][1000];
    int i,j;
    for (i=0; i < 1000; i++) {
        for (j=0; j < 1000; j++) {
            array[i][j] = 0;
        }
    }

    cout << "array[0][0] was " << array[0][0] << endl;
}
```

```
/* test2.cpp */
#include <iostream>

using namespace std;

int main() {
    int array[1000][1000];
    int i,j;
    for (i=0; i < 1000; i++) {
```

```
        for (j=0; j < 1000; j++) {
            array[j][i] = 0;
        }
    }

    cout << "array[0][0] was " << array[0][0] << endl;
}
```

```
/* test3.c */
#include <stdlib.h>

int main() {
    char* x = (char*) malloc(100);
    return 0;
}
```

```
/* test4.c */
#include <stdlib.h>

int main() {
    char* x = (char*) malloc(10);
    x[10] = 'a';
    return 0;
}
```

```
/* test5.c */
#include <stdio.h>

int main() {

    int x;
    if (x == 0) {
        printf("%s", "X is zero\n");
```

```
    }
    return 0;
}
```

Part 2

```
/* quicksort.cpp */
// Credits: https://www.geeksforgeeks.org/cpp-program-for-quicksort/
#include <bits/stdc++.h>
#include <stdlib.h>
#include <time.h>
#include <chrono>
using namespace std;

unsigned int const NUM_ELEMENTS = 50'000'000;


// A utility function to swap two elements
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // pivot
    int i = (low - 1);      // Index of smaller element and indicates
```

```
the right position of pivot found so far

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        if (arr[j] < pivot) {
            i++;  // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

/* Function to print an array */
void printArray(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
```

```
}

bool sort_verify(int arr[], int n) {
    for (int i = 1; i < n; ++i) {
        if (arr[i - 1] > arr[i]) {
            return false;
        }
    }

    return true;
}

bool quick_sort_top_level() {
    int* arr = new int[NUM_ELEMENTS];
    for (int i = 0; i < NUM_ELEMENTS; i++) {
        arr[i] = rand();
    }

    // cout << "Given array is \n";
    // printArray(arr, NUM_ELEMENTS);

    quickSort(arr, 0, NUM_ELEMENTS - 1);
    // cout << "\nSorted array is \n";
    // printArray(arr, NUM_ELEMENTS);

    bool isSorted = sort_verify(arr, NUM_ELEMENTS);
    delete[] arr;

    return isSorted;
}

// Driver Code
int main() {
    srand(time(NULL));

    int const TEST_TIMES = 5;
```

```
    int64_t test_results[TEST_TIMES];
    long long totalMilliseconds = 0;

    for (int i = 0; i < TEST_TIMES; i++) {
        auto start = std::chrono::system_clock::now();
        bool isSorted = quick_sort_top_level();
        auto end = std::chrono::system_clock::now();

        std::chrono::duration<double> elapsed = end - start;
        auto x =
std::chrono::duration_cast<chrono::milliseconds>(elapsed);
        totalMilliseconds += x.count();
        cout << "Quick Sort [" << i + 1 << "]\tsorted: " << (isSorted ?
"true" : "false") << "\tTook " << to_string(x.count()) << " ms\tn=" <<
NUM_ELEMENTS << endl;
    }

    long long average_milliseconds = (totalMilliseconds / TEST_TIMES);
    cout << "Average Milliseconds: " << average_milliseconds << " ms\n";
    cout << "Average Milliseconds per element: " <<
(double)average_milliseconds / NUM_ELEMENTS << " ms/element" << endl;

    return 0;
}
```

```
/* mergesort.cpp */
// Credits: https://www.edureka.co/blog/merge-sort-in-cpp
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <chrono>
using namespace std;
```

```
unsigned int const NUM_ELEMENTS = 50'000'000;


void merge(int a[], int Firstindex, int m, int Lastindex); //merges the
sub-arrays which are created while division

void mergeSort(int a[], int Firstindex, int Lastindex)
{
    if (Firstindex < Lastindex)
    {

        int m = Firstindex + (Lastindex - Firstindex) / 2;


        mergeSort(a, Firstindex, m);
        mergeSort(a, m + 1, Lastindex);

        merge(a, Firstindex, m, Lastindex);
    }
}

void merge(int a[], int Firstindex, int m, int Lastindex)
{
    int x;
    int y;
    int z;
    int sub1 = m - Firstindex + 1;
    int sub2 = Lastindex - m;


    int* First = new int[sub1];

    int* Second = new int[sub2];

    // copying data to temp arrays
    for (x = 0; x < sub1; x++)
```

```
{
    First[x] = a[Firstindex + x];
}
for (y = 0; y < sub2; y++)
{
    Second[y] = a[m + 1 + y];
}


x = 0;
y = 0;
z = Firstindex;
while (x < sub1 && y < sub2)
{
    if (First[x] <= Second[y])
    {
        a[z] = First[x];
        x++;
    }
    else
    {
        a[z] = Second[y];
        y++;
    }
    z++;
}
while (x < sub1)
{
    a[z] = First[x];
    x++;
    z++;
}
while (y < sub2)
{
    a[z] = Second[y];
    y++;
```

```
        z++;
    }

    delete[] First;
    delete[] Second;
}

// UTILITY FUNCTIONS
// Function to print an array
void printArray(int A[], int size) {
    for (int i = 0; i < size; i++)
        cout << A[i] << " ";
    cout << endl;
}

bool sort_verify(int arr[], int n) {
    for (int i = 1; i < n; ++i) {
        if (arr[i - 1] > arr[i]) {
            return false;
        }
    }

    return true;
}

bool merge_sort_top_level() {
    int* arr = new int[NUM_ELEMENTS];
    for (int i = 0; i < NUM_ELEMENTS; i++) {
        arr[i] = rand();
    }

    // cout << "Given array is \n";
    // printArray(arr, NUM_ELEMENTS);

    mergeSort(arr, 0, NUM_ELEMENTS - 1);
```

```
    // cout << "\nSorted array is \n";
    // printArray(arr, NUM_ELEMENTS);

    bool isSorted = sort_verify(arr, NUM_ELEMENTS);
    delete[] arr;

    return isSorted;
}

// Driver code
int main() {
    srand(time(NULL));

    int const TEST_TIMES = 5;
    int64_t test_results[TEST_TIMES];
    long long totalMilliseconds = 0;

    for (int i = 0; i < TEST_TIMES; i++) {
        auto start = std::chrono::system_clock::now();
        bool isSorted = merge_sort_top_level();
        auto end = std::chrono::system_clock::now();

        std::chrono::duration<double> elapsed = end - start;
        auto x =
std::chrono::duration_cast<chrono::milliseconds>(elapsed);
        totalMilliseconds += x.count();
        cout << "Merge Sort [" << i + 1 << "]\tsorted: " << (isSorted ?
"true" : "false") << "\tTook " << to_string(x.count()) << " ms\tn=" <<
NUM_ELEMENTS << endl;
    }

    long long average_milliseconds = (totalMilliseconds / TEST_TIMES);
    cout << "Average Milliseconds: " << average_milliseconds << " ms\n";
    cout << "Average Milliseconds per element: " <<
(double)average_milliseconds / NUM_ELEMENTS << " ms/element" << endl;
```

```
    return 0;
}
```

```
/* insertionsort.cpp */
// Credits: https://www.geeksforgeeks.org/insertion-sort/
#include <bits/stdc++.h>
#include <stdlib.h>
#include <time.h>
#include <chrono>
using namespace std;

unsigned int const NUM_ELEMENTS = 140'000;


/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            --j;
        }
        arr[j + 1] = key;
    }
}

// A utility function to print an array of size n
```

```
void printArray(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

bool sort_verify(int arr[], int n) {
    for (int i=1; i < n; ++i) {
        if (arr[i-1] > arr[i]) {
            return false;
        }
    }

    return true;
}

bool insertion_sort_top_level() {
    int* arr = new int[NUM_ELEMENTS];
    for (int i=0; i < NUM_ELEMENTS; i++) {
        arr[i] = rand();
    }

    // cout << "Given array is \n";
    // printArray(arr, NUM_ELEMENTS);

    insertionSort(arr, NUM_ELEMENTS);

    // cout << "\nSorted array is \n";
    // printArray(arr, NUM_ELEMENTS);

    bool isSorted = sort_verify(arr, NUM_ELEMENTS);
    delete[] arr;

    return isSorted;
}
```

```
/* Driver code */
int main() {
    srand(time(NULL));

    int const TEST_TIMES = 5;
    int64_t test_results[TEST_TIMES];
    long long totalMilliseconds = 0;

    for (int i = 0; i < TEST_TIMES; i++) {
        auto start = std::chrono::system_clock::now();
        bool isSorted = insertion_sort_top_level();
        auto end = std::chrono::system_clock::now();

        std::chrono::duration<double> elapsed = end - start;
        auto x =
std::chrono::duration_cast<chrono::milliseconds>(elapsed);
        totalMilliseconds += x.count();
        cout << "Insertion Sort [" << i + 1 << "]\tsorted: " <<
(isSorted ? "true" : "false") << "\tTook " << to_string(x.count()) << "
ms\tn=" << NUM_ELEMENTS << endl;
    }

    long long average_milliseconds = (totalMilliseconds / TEST_TIMES);
    cout << "Average Milliseconds: " << average_milliseconds << " ms\n";
    cout << "Average Milliseconds per element: " <<
(double)average_milliseconds / NUM_ELEMENTS << " ms/element" << endl;

}
```

Makefile

```
all: quicksort_0 mergesort_0 insertionsort_0 quicksort_3 mergesort_3
insertionsort_3

quicksort_0: quicksort.cpp
    g++ -O0 quicksort.cpp -o quicksort_0

mergesort_0: mergesort.cpp
    g++ -O0 mergesort.cpp -o mergesort_0

insertionsort_0: insertionsort.cpp
    g++ -O0 insertionsort.cpp -o insertionsort_0

quicksort_3: quicksort.cpp
    g++ -O3 quicksort.cpp -o quicksort_3

mergesort_3: mergesort.cpp
    g++ -O3 mergesort.cpp -o mergesort_3

insertionsort_3: insertionsort.cpp
    g++ -O3 insertionsort.cpp -o insertionsort_3
```

Python Script For Parsing Logs

```
import os
import pandas

# all of them start at 20 seconds
info_dict = {
    "quicksort_0": {
```

```
        "baseline": 16,  # estimate from graph number of watts
        "num": 50000000,
        "1": {
            "elapsed": 13.630,
        },
        "2": {
            "elapsed": 13.432,
        },
        "3": {
            "elapsed": 13.785,
        },
        "4": {
            "elapsed": 13.609,
        },
        "5": {
            "elapsed": 13.597,
        }
    },
    "mergesort_0": {
        "baseline": 16,
        "num": 50000000,
        "1": {
            "elapsed": 16.803,
        },
        "2": {
            "elapsed": 16.713,
        },
        "3": {
            "elapsed": 16.715,
        },
        "4": {
            "elapsed": 16.701,
        },
        "5": {
            "elapsed": 16.685,
        }
```

```
    },
    "insertionsort_0": {
        "baseline": 15,
        "num": 140000,
        "1": {
            "elapsed": 14.740,
        },
        "2": {
            "elapsed": 14.626,
        },
        "3": {
            "elapsed": 14.603,
        },
        "4": {
            "elapsed": 14.515,
        },
        "5": {
            "elapsed": 14.585,
        }
    },
    "quicksort_3": {
        "baseline": 15,
        "num": 50000000,
        "1": {
            "elapsed": 5.791,
        },
        "2": {
            "elapsed": 5.712,
        },
        "3": {
            "elapsed": 5.739,
        },
        "4": {
            "elapsed": 5.861,
        },
        "5": {
```

```
            "elapsed": 5.760,
        }
    },
    "mergesort_3": {
        "baseline": 15,
        "num": 50000000,
        "1": {
            "elapsed": 9.461,
        },
        "2": {
            "elapsed": 9.274,
        },
        "3": {
            "elapsed": 9.292,
        },
        "4": {
            "elapsed": 9.308,
        },
        "5": {
            "elapsed": 9.349,
        }
    },
    "insertionsort_3": {
        "baseline": 17,
        "num": 140000,
        "1": {
            "elapsed": 2.532,
        },
        "2": {
            "elapsed": 2.426,
        },
        "3": {
            "elapsed": 2.444,
        },
        "4": {
            "elapsed": 2.430,
```

```
        },
        "5": {
            "elapsed": 2.415,
        }
    },
}


rootdir = "logs"
for dirpath, dirnames, filenames in os.walk(rootdir):
    curfolder = os.path.basename(dirpath)
    if curfolder.startswith("insertion") or
curfolder.startswith("quick") or curfolder.startswith("merge"):
        print(curfolder)
        package_power_filename = os.path.join(
            rootdir, curfolder, "[-]", "[CSV]", "Intel Core i7 7700HQ
Package Power.csv")
        if os.path.exists(package_power_filename):
            data = pandas.read_csv(package_power_filename, header=None,
                    usecols=[0, 1], names=['Time', "Watts"])

            sorting_start_time = 20

            # sorting data is part of data where time is past 20 seconds
            sorting_data = data.loc[data['Time'] >= sorting_start_time]

            # folder name is used as key to info_dict
            algodict = info_dict[curfolder]
            cur_start_time = sorting_start_time
            algo_rounds_avg_joules = []

            # looping through each algorithm information
            for i in range(1, 6):
                # looping through each round
                round_elapsed_time = algodict[str(i)]["elapsed"]
                round_data = sorting_data.loc[(sorting_data['Time'] >=
```

```
cur_start_time) & (sorting_data['Time'] < (cur_start_time +
round_elapsed_time))]
                cur_start_time += round_elapsed_time

                round_avg_watts = round_data["Watts"].mean()
                if round_data["Watts"].empty:
                    print(sorting_data['Time'])
                    input(f"no it is empty from {cur_start_time} to
{(cur_start_time + round_elapsed_time)}")
                # print(f"{curfolder}
[{i}]:\tavg_watts={round_avg_watts:.3f}\telapsed_seconds={round_elapsed_
time}\tavg_joules={round_avg_watts * round_elapsed_time:.3f}")
                print(f"{round_avg_watts * round_elapsed_time:.3f}")
                algo_rounds_avg_joules.append(round_avg_watts *
round_elapsed_time)

            algo_avg_joules = sum(algo_rounds_avg_joules) /
len(algo_rounds_avg_joules)
            num_elements = algodict["num"]
            print(f"{curfolder}
total:\tavg_joules={algo_avg_joules:.3f}\tavg_joules/element={algo_avg_j
oules/num_elements:.6f}\n")
```