CM2303
Run-Time Analysis of Insertion Sort and Counting Sort Algorithms

David Buchanan, C1673152

February 6, 2018

Contents

1 Introduction

The aim of this report is to compare the theoretical run-time with the real-world run-time of the insertion sort and counting sort algorithms. I will do this by measuring the performance of each algorithm with a wide range of input data.

By using a highly accurate timing mechanism with nanosecond-scale resolution, I aim to be able to detect discrepancies between the theoretical and real-world run-times of the algorithms on an Intel x86-64 PC platform. Such discrepancies could arise from branch prediction accuracy, CPU cache management, CPU power management, and other subtle micro-architectural features.

The theoretical run-time complexities of the two algorithms are as follows:

Algorithm	Worst Case	Average Case	Best Case
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Counting Sort	$O(n)$	$O(n)$	$O(n)$

For insertion sort, a best-case input would be already-sorted data, resulting in $O(n)$ time complexity, since the inner loop will never execute. Conversely, the worst-case input would be data sorted in reverse order, resulting in $O(n^2)$ time complexity. Although this is the same time complexity as the average case, it should be a constant factor slower, because the inner loop will execute a greater number of times on average.

For counting sort, the theoretical time complexity is the same for best, worst, and average cases. However, the runtime also depends on a constant k, where k is the number of possible data values i.e. the actual run-time is bounded by $O(n+k)$. If k is large compared to n, it will have a big impact on the run-time of the algorithm.

2 Algorithms

2.1 Pseudocode

2.1.1 Insertion Sort

Algorithm InsertionSort(A, n)
Input: an array A storing n integers
Output: array A sorted in non–descending order
for $i \leftarrow 1$ **to** $n-1$ **do**
  $item \leftarrow A[i]$
  $j \leftarrow i-1$
  while $j \geq 0$ **and** $A[j] > item$ **do**
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j-1$
  end while
  $A[j+1] \leftarrow item$
end for

2.1.2 Counting Sort

Algorithm CountingSort(A, B, n, k)
Input: array A with n elements, each with value from 0 to $k-1$
Output: sorted array B
for $i \leftarrow 0$ **to** $k-1$ **do**
  $C[i] \leftarrow 0$
end for
for $j \leftarrow 0$ **to** $n-1$ **do**
  $C[A[j]] \leftarrow C[A[j]]+1$
end for
for $i \leftarrow 1$ **to** $k-1$ **do**
  $C[i] \leftarrow C[i]+C[i+1]$
end for
for $j \leftarrow n-1$ **downto** 0 **do**
  $B[C[A[j]]-1] \leftarrow A[j]$
  $C[A[j]] \leftarrow C[A[j]]-1$
end for

2.2 Implementation

I wanted to make sure that my timings were as precise and exact as accurate as possible. I took several steps to ensure this. Firstly, I executed my algorithms inside a Linux Kernel Module. This provides multiple benefits:

- I can seize control of the CPU by disabling preemption and hardware interrupts, making my code uninterruptible by other tasks.
- I can use privileged CPU instructions to flush the CPU caches prior to each test, to ensure fairness. I can also experiment with disabling the CPU cache entirely, for example.

Secondly, I used inline assembly code to access the TSC (Time Stamp Counter) register, to measure time. This register essentially measures individual CPU clock cycles, and is the most accurate monotonic time source available on the x86 PC platform. This technique is based on the one documented by Gabriele Paoloni[1].

To further improve the accuracy of my results, I disabled CPU frequency scaling and multi-core processing on the system on which I ran the benchmarks.

The kernel module uses the `ioctl` interface in order to make its benchmarking functionality accessible to user-mode code. I used a Python script to make calls into my kernel module via this interface, and record the results in `.csv` files. I used a second Python script to process the raw results into graphs.

2.2.1 Linux Kernel Module

kmod/benchmark.c

```c
#define LINUX

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/vmalloc.h>
#include <linux/random.h>
#include <linux/device.h>
#include <linux/uaccess.h>

#define ELEM_TYPE u64
#include "algorithms.h"

#define DRIVER_AUTHOR "David Buchanan"
#define DRIVER_DESC   "Pointlessly precise benchmarking"
#define DEVICE_NAME   "benchmark"
#define DEVICE_MAJOR  100 // this should not really be hardcoded

#define CR0_CD (1<<30)

#define N_MAX 0x80000 // arbitrary limit on maximum N value
#define IOCTL_BENCH_ISORT 0
#define IOCTL_BENCH_CSORT 1
#define ISORT_MODE_RANDOM 0
#define ISORT_MODE_ASCENDING 1
#define ISORT_MODE_DESCENDING 2

ELEM_TYPE * randcache;

struct ioctl_arg {
    u64 n; // n is reused to store the result
    u64 k; // k is reused by the insertion sort handler to chose the type of data
    u64 cache_enabled;
};

static inline void set_cache(int enabled)
{
    u64 cr0tmp;

```

```
    /* read the cr0 register into a variable */
    asm volatile (
        "mov %%cr0, %0;"
        : "=r" (cr0tmp)
    );

    if (enabled) {
        cr0tmp &= ~CR0_CD; // clear the Cache Disable bit
    } else {
        cr0tmp |= CR0_CD; // set the Cache Disable bit
    }

    /* The cache must be invalidated after it is disabled
    in order to maintain cache coherency */
    asm volatile (
        "mov %0, %%cr0;"
        "wbinvd;"
        :
        : "r" (cr0tmp)
    );
}

static inline void flush_cache(void)
{
    asm volatile ("wbinvd;");
}

volatile u64 bench_isort(size_t n, int mode, int cache_enabled)
{
    ELEM_TYPE * a;
    size_t i;
    unsigned long flags;
    u64 t0, t1;
    u32 t0lo, t0hi, t1lo, t1hi;

    printk("bench_isort(%lu, %d)\n", n, cache_enabled);

    a = vmalloc(n * sizeof(*a));

    switch (mode) {
        case ISORT_MODE_RANDOM: // average case
            for (i = 0; i < n; i++) {
                a[i] = randcache[i];
            }
            break;
        case ISORT_MODE_ASCENDING: // best case
            for (i = 0; i < n; i++) {
                a[i] = i;
            }
            break;
        case ISORT_MODE_DESCENDING: // worst case
            for (i = 0; i < n; i++) {
                a[i] = n-i-1;
            }
            break;
        default:
            return -1;
    }

    preempt_disable();
    raw_local_irq_save(flags);
```

```c
    flush_cache();

    if (!cache_enabled) set_cache(0); // disable CPU cache

    asm    volatile (
        "cpuid;"
        "rdtsc;"
        "mov %%edx, %0;"
        "mov %%eax, %1;"
        : "=r" (t0hi), "=r" (t0lo)
        :
        : "rax", "rbx", "rcx", "rdx"
    );

    isort(a, n); // the compiler should inline this

    asm    volatile (
        "rdtscp;"
        "mov %%edx, %0;"
        "mov %%eax, %1;"
        "cpuid;"
        : "=r" (t1hi), "=r" (t1lo)
        :
        : "rax", "rbx", "rcx", "rdx"
    );

    if (!cache_enabled) set_cache(1); // reenable CPU cache

    raw_local_irq_restore(flags);
    preempt_enable();

    vfree(a);

    t0 = ((u64) t0hi << 32) | t0lo;
    t1 = ((u64) t1hi << 32) | t1lo;

    return t1-t0;
}

volatile u64 bench_csort(size_t n, ELEM_TYPE k, int cache_enabled)
{
    ELEM_TYPE * a, * b;
    size_t * c, i;
    unsigned long flags;
    u64 t0, t1;
    u32 t0lo, t0hi, t1lo, t1hi;

    printk("bench_csort(%lu, %llu, %d)\n", n, k, cache_enabled);

    a = vmalloc(n * sizeof(*a));
    b = vmalloc(n * sizeof(*b));
    c = vmalloc(k * sizeof(*c));

    for (i = 0; i < n; i++) {
        a[i] = randcache[i] % k;
    }

    preempt_disable();
    raw_local_irq_save(flags);

```

```
    flush_cache();

    if (!cache_enabled) set_cache(0); // disable CPU cache

    asm    volatile (
        "cpuid;"
        "rdtsc;"
        "mov %%edx, %0;"
        "mov %%eax, %1;"
        : "=r" (t0hi), "=r" (t0lo)
        :
        : "rax", "rbx", "rcx", "rdx"
    );

    csort(a, b, c, n, k); // the compiler should inline this

    asm    volatile (
        "rdtscp;"
        "mov %%edx, %0;"
        "mov %%eax, %1;"
        "cpuid;"
        : "=r" (t1hi), "=r" (t1lo)
        :
        : "rax", "rbx", "rcx", "rdx"
    );

    if (!cache_enabled) set_cache(1); // reenable CPU cache

    raw_local_irq_restore(flags);
    preempt_enable();

    vfree(a);
    vfree(b);
    vfree(c);

    t0 = ((u64) t0hi << 32) | t0lo;
    t1 = ((u64) t1hi << 32) | t1lo;

    return t1-t0;
}

/* stub */
static int device_open(struct inode *inode, struct file *file)
{
    return 0;
}

/* stub */
static int device_release(struct inode *inode, struct file *file)
{
    return 0;
}

/* stub */
static ssize_t device_read(
    struct file *f,
    char __user *buf,
    size_t len,
    loff_t *off)
{
    return 0;
```

```c
}

/* stub */
static ssize_t device_write(
    struct file *f,
    const char __user *buf,
    size_t len,
    loff_t *off)
{
    return len;
}

long device_ioctl(
    struct file *file,
    unsigned int ioctl_num,/* The number of the ioctl */
    unsigned long ioctl_param) /* The parameter to it */
{
    struct ioctl_arg args;

    if (copy_from_user(&args, (void *) ioctl_param, sizeof(args)) != 0) {
        return -EACCES;
    }

    switch (ioctl_num) {
        case IOCTL_BENCH_ISORT:
            args.n = bench_isort(args.n, args.k, args.cache_enabled);

            if (copy_to_user((void *) ioctl_param, &args, sizeof(args)) != 0) {
                return -EACCES;
            }

            return 0;

        case IOCTL_BENCH_CSORT:
            if (args.n > N_MAX) {
                return -1;
            }

            args.n = bench_csort(args.n, args.k, args.cache_enabled);

            if (copy_to_user((void *) ioctl_param, &args, sizeof(args)) != 0) {
                return -EACCES;
            }

            return 0;
    }
    return -1;
}

const struct file_operations fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = device_ioctl,
    .open = device_open,
    .release = device_release,
    .read = device_read,
    .write = device_write
};

int init_module(void)
{
    int result;
```

```c
    result = register_chrdev(DEVICE_MAJOR, DEVICE_NAME, &fops);

    if (result < 0) {
        return result;
    }

    /* initialse a large cache of random data to speed things up */
    /* Generating random data each time would be slow  */
    randcache = vmalloc(sizeof(randcache) * N_MAX);
    get_random_bytes(randcache, sizeof(randcache) * N_MAX);

    printk("Benchmarker loaded.\n");

    return 0;
}

void cleanup_module(void)
{
    printk("Benchmarker unloading\n");
    return unregister_chrdev(DEVICE_MAJOR, DEVICE_NAME);
}

MODULE_LICENSE("Dual MIT/GPL");

MODULE_AUTHOR(DRIVER_AUTHOR);
MODULE_DESCRIPTION(DRIVER_DESC);
```

kmod/algorithms.h

```c
#ifndef __HAVE_ARCH_MEMSET
#include <string.h>
#endif

/* used for debugging */
#ifdef TESTING
void print_array(const char * name, ELEM_TYPE * array, size_t length)
{
    printf("%s = {", name);
    for (int i = 0; i < length; i++) {
        printf("%u, ", array[i]);
    }
    printf("\b\b}\n");
}
#endif

void isort(ELEM_TYPE * a, size_t n)
{
    typeof(*a) tmp;
    typeof(n) i, j;

    for (i = 1; i < n; i++) {
        tmp = a[i];
        for (j = i; j-- > 0 && a[j] > tmp;) {
            a[j+1] = a[j];
        }
        a[j+1] = tmp;
#ifdef TESTING
        printf("i = %u, ", i);
        print_array("a", a, n);
#endif
    }
}
```

```c

static inline void csort(
    ELEM_TYPE * a, /* input array */
    ELEM_TYPE * b, /* output array */
    size_t * c, /* count array */
    size_t n, /* number of elements in a (and therfore b) */
    ELEM_TYPE k) /* number of elements in c (upper limit of values in a) */
{
    typeof(n) j;
    typeof(k) i;

    /* idiomatic implementation of first loop from pseudocode */
    memset(c, 0, k * sizeof(*c));

    for (j = 0; j < n; j++) c[a[j]]++;

#ifdef TESTING
    print_array("c after 2nd loop", c, k);
#endif

    for (i = 1; i < k; i++) c[i] += c[i-1];

#ifdef TESTING
    print_array("c after 3rd loop", c, k);
#endif

    for (j = n; j-- > 0; ) {
        b[c[a[j]] - 1] = a[j];
        c[a[j]]--;
    }
}
```

2.2.2 Python Client

client/client.py

```python
import struct
import fcntl
import math

# this device must be created first via `mknod /dev/benchmark c 100 0`
BENCH_DEVICE = "/dev/benchmark"

IOCTL_BENCH_ISORT = 0
IOCTL_BENCH_CSORT = 1

ISORT_MODE_RANDOM = 0
ISORT_MODE_ASCENDING = 1
ISORT_MODE_DESCENDING = 2

STEP = 256

def run_bench(device, ioctl_no, n, k=0, cache_enabled=1):
    args = struct.pack("<QQQ", n, k, cache_enabled)
    result = fcntl.ioctl(device, ioctl_no, args, False)
    return struct.unpack_from("<Q", result)[0]

with open(BENCH_DEVICE) as b:
    # COUNTING SORT, CACHE ENABLED, K=N
    with open("../results/csort_cache_n.csv", "w") as outfile:
        for i in range(1, 0x40000, STEP):
            time = run_bench(b, IOCTL_BENCH_CSORT, i, i, 1)
```

```
            outfile.write("{},\t{}\n".format(i, time))
            print(i)

    # COUNTING SORT, NO CACHE, K=N
    with open("../results/csort_nocache_n.csv", "w") as outfile:
        for i in range(1, 0x8000, STEP):
            time = run_bench(b, IOCTL_BENCH_CSORT, i, i, 0)
            outfile.write("{},\t{}\n".format(i, time))
            print(i)

    # COUNTING SORT, NO CACHE, K=2N
    with open("../results/csort_nocache_2n.csv", "w") as outfile:
        for i in range(1, 0x8000, STEP):
            time = run_bench(b, IOCTL_BENCH_CSORT, i, 2*i, 0)
            outfile.write("{},\t{}\n".format(i, time))
            print(i)

    # COUNTING SORT, NO CACHE, K=1
    with open("../results/csort_nocache_1.csv", "w") as outfile:
        for i in range(1, 0x8000, STEP):
            time = run_bench(b, IOCTL_BENCH_CSORT, i, 1, 0)
            outfile.write("{},\t{}\n".format(i, time))
            print(i)

    # COUNTING SORT, NO CACHE, K=50000
    with open("../results/csort_nocache_50000.csv", "w") as outfile:
        for i in range(1, 0x8000, STEP):
            time = run_bench(b, IOCTL_BENCH_CSORT, i, 50000, 0)
            outfile.write("{},\t{}\n".format(i, time))
            print(i)

    # COUNTING SORT, CACHE ENABLED, RANDOM ORDER
    with open("../results/isort_cache_random.csv", "w") as outfile:
        for i in range(1, 0x20000000, STEP*8192*2):
            n = int(math.sqrt(i))
            time = run_bench(b, IOCTL_BENCH_ISORT, n, ISORT_MODE_RANDOM)
            outfile.write("{},\t{}\n".format(n, time))
            print(n)

    # COUNTING SORT, CACHE ENABLED, ASCENDING ORDER
    with open("../results/isort_cache_ascending.csv", "w") as outfile:
        for i in range(1, 0x20000000, STEP*8192*2):
            n = int(math.sqrt(i))
            time = run_bench(b, IOCTL_BENCH_ISORT, n, ISORT_MODE_ASCENDING)
            outfile.write("{},\t{}\n".format(n, time))
            print(n)

    # COUNTING SORT, CACHE ENABLED, DESCENDING ORDER
    with open("../results/isort_cache_descending.csv", "w") as outfile:
        for i in range(1, 0x20000000, STEP*8192*2):
            n = int(math.sqrt(i))
            time = run_bench(b, IOCTL_BENCH_ISORT, n, ISORT_MODE_DESCENDING)
            outfile.write("{},\t{}\n".format(n, time))
            print(n)

    # COUNTING SORT, CACHE DISABLED, ASCENDING ORDER
    with open("../results/isort_nocache_ascending.csv", "w") as outfile:
        for i in range(1, 0x8000, STEP):
            time = run_bench(b, IOCTL_BENCH_ISORT, i, ISORT_MODE_ASCENDING, 0)
            outfile.write("{},\t{}\n".format(i, time))
            print(i)
```

2.2.3 Python Plotting

client/graph.py

```
import matplotlib.pyplot as plt
import numpy as np

SCALE=(8,5)

def read_data(filename):
    return [(int(n.strip()) for n in line.split(",")) for line in open(filename).read().split("\n") if

def print_10_points(data):
    print("\\begin{center}")
    print("\\begin{tabular}{ |c|c| } ")
    print("\\hline")
    print("n & clock cycles \\\\")
    print("\\hline")
    num_points = len(x)
    for i in range(0, num_points, num_points//10):
        print("{} \\\\".format(" & ".join([str(n) for n in data[i]])))
    print("\\hline")
    print("\\end{tabular}")
    print("\\end{center}")

x, y = zip(*read_data("../results/csort_cache_n.csv"))

print("COUNTING CACHE")
print_10_points(list(zip(x, y)))

plt.figure(figsize=SCALE)
plt.scatter(x, y, s=4, linewidth=0.1, c="k", marker="x")
plt.ylabel("clock cycles")
plt.xlabel("n")
plt.axvline(x=(3096*1024)/(8*3), c="r", ls="--", label="memory needed = L3 cache size")
plt.axvline(x=(512*1024)/(8*3), c="g", ls="--", label="memory needed = L2 cache size")
plt.axvline(x=(128*1024)/(8*3), ls="--", label="memory needed = L1 cache size")
plt.legend()
plt.grid()
plt.savefig("../report/plots/csort_cache_n.svg")
plt.show()

print("COUNTING NOCACHE")
plt.figure(figsize=SCALE)
ys = []
for filename in ["n", "2n", "1", "50000"]:
    x, y = zip(*read_data("../results/csort_nocache_{}.csv".format(filename)))
    ys.append(y)
    coef = np.corrcoef(x, y)[1][0]
    plt.scatter(x, y, s=10, linewidth=1, marker="x", label="k = {}      (Correlation coefficient {:.6f})
print_10_points(list(zip(*([x]+ys))))

plt.ylabel("clock cycles")
plt.xlabel("n")
plt.legend()
plt.grid()
plt.savefig("../report/plots/csort_nocache.svg")
plt.show()

print("INSERTION CACHE")
plt.figure(figsize=SCALE)
ys = []
```

```
for filename in ["random", "ascending", "descending"]:
    x, y = zip(*read_data("../results/isort_cache_{}.csv".format(filename)))
    x = [n*n for n in x]
    ys.append(y)
    coef = np.corrcoef(x, y)[1][0]
    plt.scatter(x, y, s=10, linewidth=1, marker="x", label="{} order (Correlation coefficient {:.6f})".
print_10_points(list(zip(*([x]+ys))))

plt.ylabel("clock cycles")
plt.xlabel("n\u00B2")
plt.legend()
plt.grid()
plt.savefig("../report/plots/isort_cache.svg")
plt.show()

print("INSERTION NOCACHE")
plt.figure(figsize=SCALE)
x, y = zip(*read_data("../results/isort_nocache_ascending.csv"))
print_10_points(list(zip(x, y)))
coef = np.corrcoef(x, y)[1][0]
plt.scatter(x, y, s=10, linewidth=1, marker="x", label="ascending order (Correlation coefficient {:.6f}

plt.ylabel("clock cycles")
plt.xlabel("n")
plt.legend()
plt.grid()
plt.savefig("../report/plots/isort_nocache_ascending.svg")
plt.show()
```

3 Testing

I wrote two small test programs in C in order to verify the correctness of my algorithm implementations. These test programs run in user mode for convenience, but still use the same algorithm code as the kernel module. By defining the TESTING preprocessor macro, I enabled print statements within the algorithms to display intermediate array and variable values, where appropriate. The source code for the tests is as follows:

tests/test_isort.c

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

#define TESTING
#define ELEM_TYPE uint64_t
#include "../kmod/algorithms.h"

#define TEST(array) \
    puts("\nTesting dataset '" #array "':"); \
    print_array(#array, array, sizeof(array)/sizeof(*array)); \
    isort(array, sizeof(array)/sizeof(*array)); \
    print_array("final " #array, array, sizeof(array)/sizeof(*array));

ELEM_TYPE test1[] = {5, 4, 3, 2, 1}; // reverse sorted
ELEM_TYPE test2[] = {1, 2, 3, 4, 5}; // sorted
ELEM_TYPE test3[] = {7, 1, 0, 5, 9, 2, 7, 1}; // random order with duplicates

int main(int argc, char * argv[])
{
    TEST(test1);
    TEST(test2);
    TEST(test3);
}
```

tests/test_csort.c

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

#define TESTING
#define ELEM_TYPE uint64_t
#include "../kmod/algorithms.h"

// For all tests, k will be 10
#define K 10

#define TEST(array, array_out) \
    puts("\nTesting dataset '" #array "':"); \
    print_array(#array, array, sizeof(array)/sizeof(*array)); \
    csort(array, array_out, counts, sizeof(array)/sizeof(*array), K); \
    print_array(#array_out, array_out, sizeof(array)/sizeof(*array));

ELEM_TYPE test1[] = {5, 4, 3, 2, 1}; // reverse sorted
ELEM_TYPE test2[] = {1, 2, 3, 4, 5}; // sorted
ELEM_TYPE test3[] = {7, 1, 0, 5, 9, 2, 7, 1}; // random order with duplicates

int main(int argc, char * argv[])
{
    ELEM_TYPE * test1_sorted = malloc(sizeof(test1));
    ELEM_TYPE * test2_sorted = malloc(sizeof(test2));
    ELEM_TYPE * test3_sorted = malloc(sizeof(test3));

    size_t * counts = malloc(K * sizeof(*counts)); // reused for each test

    TEST(test1, test1_sorted);
    TEST(test2, test2_sorted);
    TEST(test3, test3_sorted);

    free(test1_sorted);
    free(test2_sorted);
    free(test3_sorted);
    free(counts);
}
```

3.1 Test Results

When compiled and executed, they produced the following outputs respectively:

3.1.1 Insertion Sort

```
Testing dataset 'test1':
test1 = {5, 4, 3, 2, 1}
i = 1, a = {4, 5, 3, 2, 1}
i = 2, a = {3, 4, 5, 2, 1}
i = 3, a = {2, 3, 4, 5, 1}
i = 4, a = {1, 2, 3, 4, 5}
final test1 = {1, 2, 3, 4, 5}

Testing dataset 'test2':
test2 = {1, 2, 3, 4, 5}
i = 1, a = {1, 2, 3, 4, 5}
i = 2, a = {1, 2, 3, 4, 5}
i = 3, a = {1, 2, 3, 4, 5}
i = 4, a = {1, 2, 3, 4, 5}
final test2 = {1, 2, 3, 4, 5}
```

```
Testing dataset 'test3':
test3 = {7, 1, 0, 5, 9, 2, 7, 1}
i = 1, a = {1, 7, 0, 5, 9, 2, 7, 1}
i = 2, a = {0, 1, 7, 5, 9, 2, 7, 1}
i = 3, a = {0, 1, 5, 7, 9, 2, 7, 1}
i = 4, a = {0, 1, 5, 7, 9, 2, 7, 1}
i = 5, a = {0, 1, 2, 5, 7, 9, 7, 1}
i = 6, a = {0, 1, 2, 5, 7, 7, 9, 1}
i = 7, a = {0, 1, 1, 2, 5, 7, 7, 9}
final test3 = {0, 1, 1, 2, 5, 7, 7, 9}
```

3.1.2 Counting Sort

```
Testing dataset 'test1':
test1 = {5, 4, 3, 2, 1}
c after 2nd loop = {0, 1, 1, 1, 1, 1, 0, 0, 0, 0}
c after 3rd loop = {0, 1, 2, 3, 4, 5, 5, 5, 5, 5}
test1_sorted = {1, 2, 3, 4, 5}

Testing dataset 'test2':
test2 = {1, 2, 3, 4, 5}
c after 2nd loop = {0, 1, 1, 1, 1, 1, 0, 0, 0, 0}
c after 3rd loop = {0, 1, 2, 3, 4, 5, 5, 5, 5, 5}
test2_sorted = {1, 2, 3, 4, 5}

Testing dataset 'test3':
test3 = {7, 1, 0, 5, 9, 2, 7, 1}
c after 2nd loop = {1, 2, 1, 0, 0, 1, 0, 2, 0, 1}
c after 3rd loop = {1, 3, 4, 4, 4, 5, 5, 7, 7, 8}
test3_sorted = {0, 1, 1, 2, 5, 7, 7, 9}
```

4 Experimental Setup and Results

4.1 Results

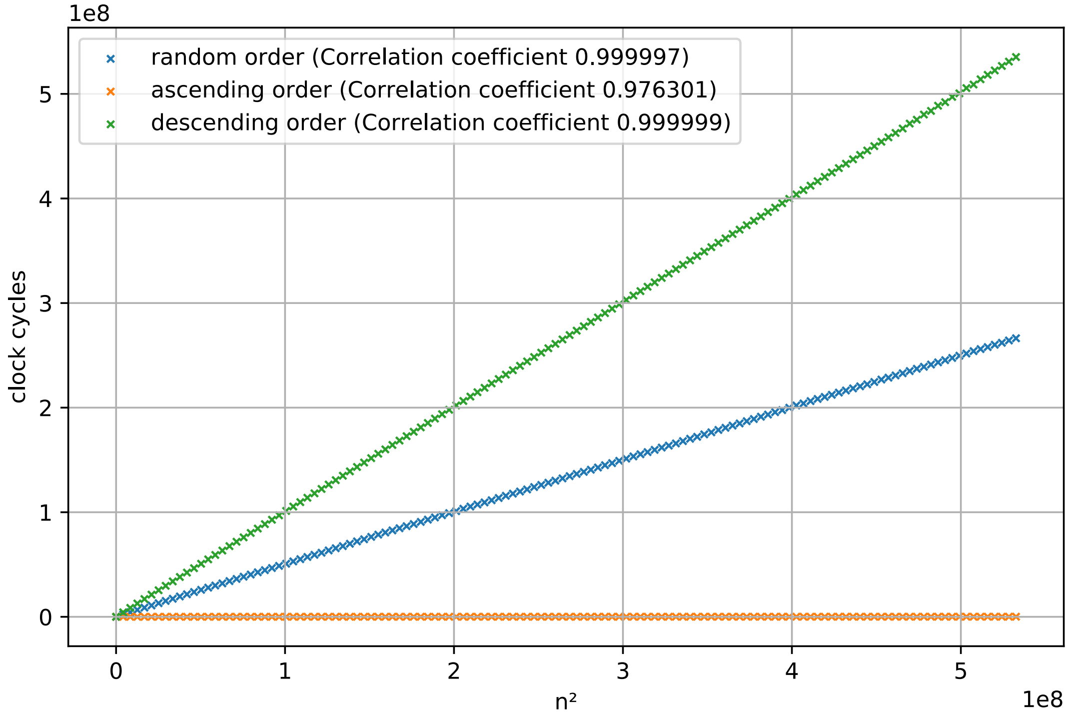
Overall, the benchmarks took only a few minutes to execute. Note that all timings are recorded in CPU clock cycles. The actual duration of a clock cycles is irrelevant to this analysis, since it is a constant factor, however it is still useful to know for context. The CPU used to collect these results was an Intel Core i3 2310M, running at 2.1GHz. Therefore, each clock cycle is approximately 476 Picoseconds in duration. According to the Intel datasheets, it has a 3MiB L3 cache, a 512KiB L2 cache and a 128KiB L1 cache. The relevance of these details will be explained later in my conclusion. The system was running Ubuntu Linux 17.10 x86_64, kernel version 4.13.0.

All graphs are rendered as scatter plots. The correlation coefficients quoted were calculated as Pearson product-moment correlation coefficients, where applicable.

Due to the apparent accuracy of my timing mechanism, I decided that there was no need to take any repeat measurements, however doing so could have slightly increased the accuracy of my results.

Where result tables are shown, these contain only a subset of the data used to plot the graphs. The full datasets have been submitted in `.csv` files along with this report.

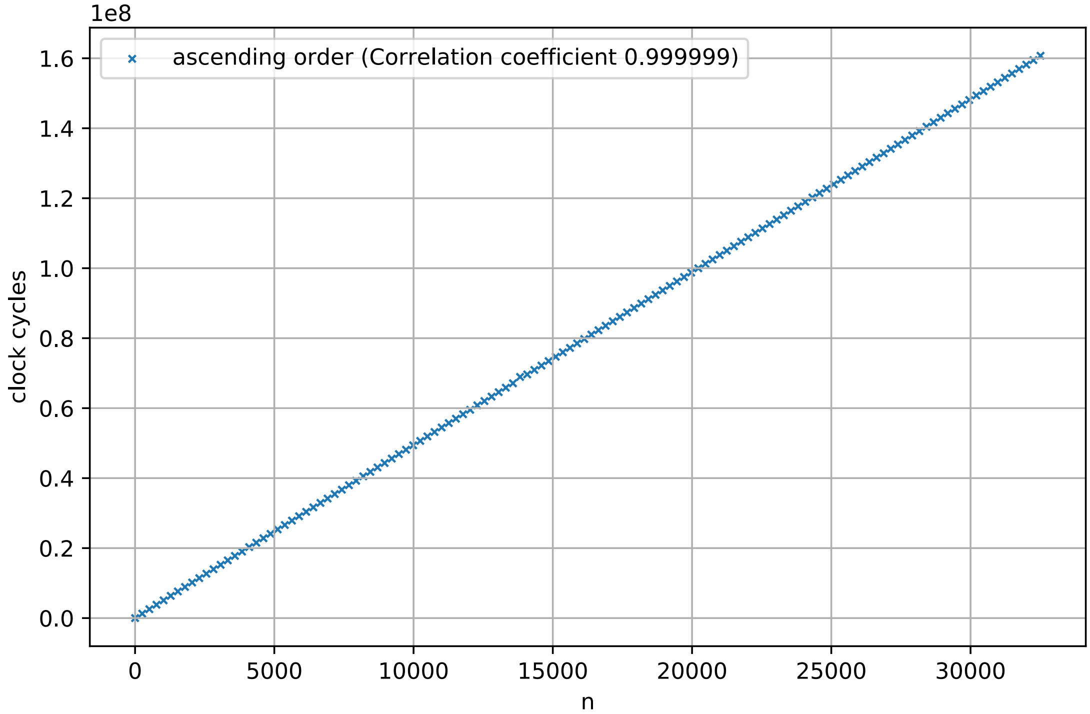
4.1.1 Insertion Sort (Best, Worst, and Average cases) - CPU Cache Enabled



n^2	random order (clock cycles)	ascending order (clock cycles)	descending order (clock cycles)
1	260	264	264
50324836	25820436	30808	50726824
100661089	50741164	43644	101245916
150994944	76445616	53112	151723488
201299344	101071480	61108	202143968
251634769	126115408	68008	252582924
301960129	151263080	75624	303008584
352312900	176424100	81172	353448744
402644356	202084732	85696	403861888
452966089	226860224	91552	454259212
503284356	251737844	96776	505555068

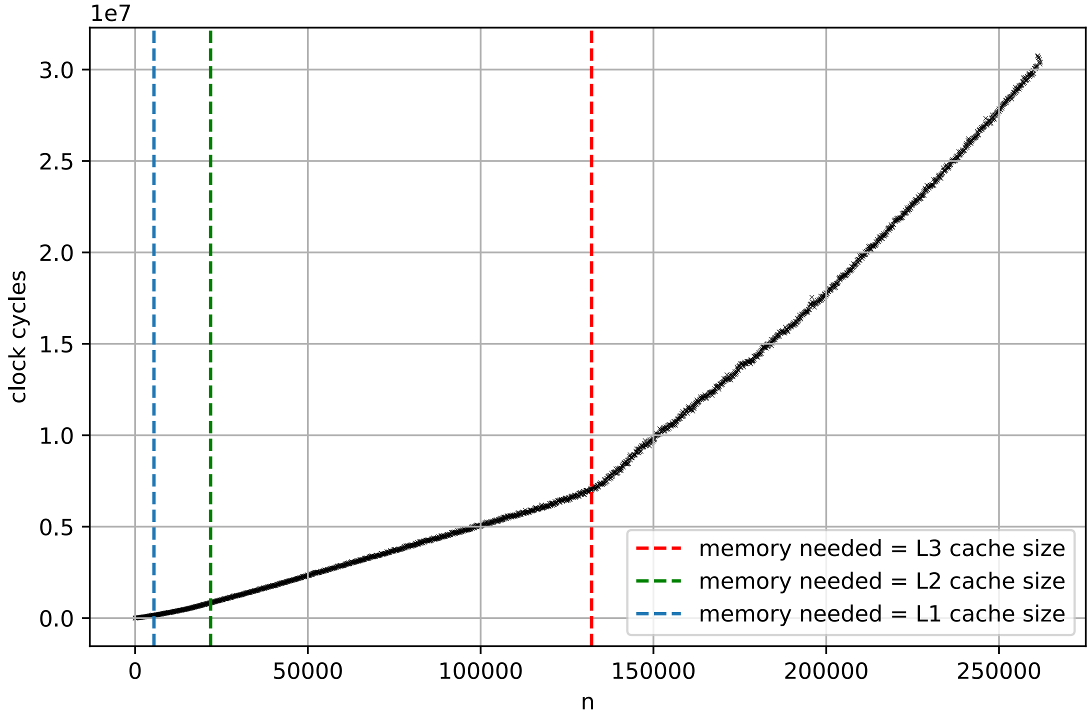
Side note: for $n = 1$, the run-times were all measured to be within 4 clock cycles of each other, less than 2 nanoseconds! Since an array of length 1 is already sorted, identical results are expected for $n = 1$.

4.1.2 Insertion Sort Best Case (Linear x Axis) - CPU Cache Disabled



n	clock cycles
1	6684
3073	15229484
6145	30375664
9217	45590940
12289	60806388
15361	75966808
18433	91154052
21505	106290700
24577	121489108
27649	136656552
30721	151875412

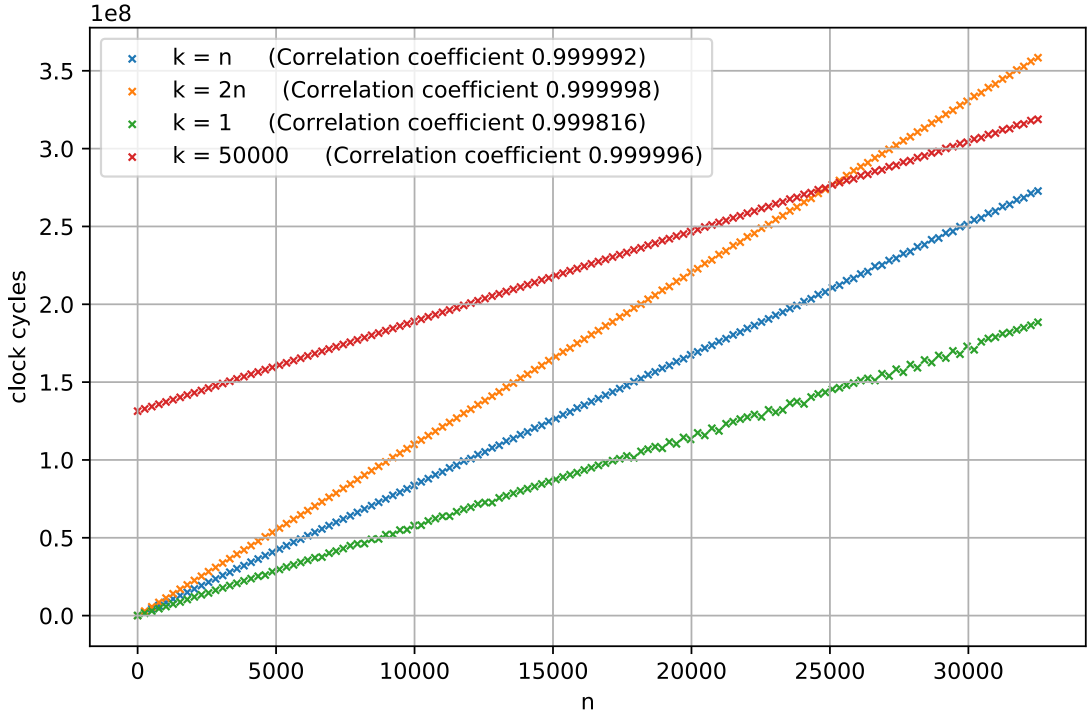
4.1.3 Counting Sort ($k = n$) – CPU Cache Enabled



Note: This is also a scatter plot! Zoom in if you want to see the individual samples.

n	clock cycles
1	752
26113	1055240
52225	2505096
78337	3950100
104449	5385020
130561	6985208
156673	10715304
182785	14976496
208897	19574708
235009	24676932
261121	30739212

4.1.4 Counting Sort with Various k Values - CPU Cache Disabled



n	$k = n$	$k = 2n$	$k = 1$	$k = 50000$
1	23240	27816	23628	131327780
3073	25803272	33786356	17793940	149121876
6145	51310344	67441884	35586820	166837064
9217	77262460	101548652	52187272	184478744
12289	103413348	135492768	71703516	202243804
15361	129043204	169368768	89037280	219980132
18433	154806644	203235232	106899120	237932104
21505	180335944	237689800	124554720	255464776
24577	206230420	271259704	142383476	273585548
27649	232485524	305071932	156358572	291259676
30721	258439040	339196328	178022744	309001932

5 Conclusion

The only graph which stands out as unusual is in section 4.1.3, the Counting Sort with CPU cache enabled. The graph starts off relatively linear, but then the gradient increases sharply at approximately $n = 132096$. This is no coincidence – It is the point at which the data required for the algorithm no longer fits entirely in the CPU's 3MiB L3 cache (The A, B, and C arrays each have 8-byte sized elements, $132096 \times 8 \times 3 = 3 \times 2^{20} = 3$MiB). When not all the data can fit in the cache, the probability that the CPU will need to fetch a section of memory from the system DRAM rather than the cache increases (the cache miss ratio), which incurs a performance penalty. The same effect occurs when crossing the L1 and L2 cache size boundaries, but it is smaller in magnitude and less noticeable.

To mitigate the effects of the somewhat unpredictable data caching, I ran the rest of the counting sort benchmarks with the CPU's cache disabled, shown in section 4.1.4 (This incurs a very heavy performance penalty, increasing run-times by an order of magnitude). However, the CPU cache hierarchy did not seem to affect the results for Insertion Sort, and I believe this is because all memory accesses are purely sequential, so the CPU's internal cache prefetch engine is always able to have the relevant memory in the cache before it is needed. On the other hand, memory accesses for most of Counting sort are random in order (assuming random input data), so the CPU is unable to prefetch memory into the caches effectively.

The results for Insertion Sort (section 4.1.1) perfectly match the theoretical run-time complexities for the best, worst and average cases. The time complexities for the Worst and Average case (descending order and random order) are both clearly $O(n^2)$, with a correlation coefficient greater than 0.99999. As predicted, the worst-case times are approximately double the average-case times. The best case (ascending order) has a run-time complexity of $O(n)$, and as such it is not easy to see the details of with n^2 as the x-axis. Therefore I made a separate plot showing just the best-case run-times (section 4.1.2), which confirms that the real-world run-time complexity is $O(n)$, with a correlation coefficient of over 0.999999.

The results for Counting Sort (section 4.1.4) also match the theoretical run-time complexity of $O(n)$. If we compare the cases where $k = 1$ and $k = 50000$, we can see that the two graphs have the same gradient, but where $k = 500000$, the graph is shifted upwards on the y-axis. Comparing the cases where $k = 1$, $k = n$ and $k = 2n$, we can see that if k is a multiple of n, then the gradient of the graph is steeper as the n coefficient increases. For some reason, the case where $k = 1$ has a slightly lower correlation coefficient of 0.999816, corresponding to slightly more variance in the measured times. I am unable to explain this variation (I re-ran the benchmarks several times). Overall, this confirms my prediction that a larger value of k results in worse performance.

5.1 Advice

Clearly, counting sort is superior in terms of time complexity, *however* in most real-world cases the value of k would be very large. For example, when sorting an array of 64-bit integers, $k = 2^{64}$. That would take an incomprehensible amount of time to compute, but also at least 2^{64} bytes of memory for the C array. This is obviously completely infeasible. On the other hand if the value of n is very large, then Insertion Sort may be infeasible in terms of time. This leads on to an advantage of Insertion Sort – It sorts data entirely in-place without requiring any additional memory. The sequential memory access patterns also allow the CPU to utilise it's hardware cache prefetch mechanisms effectively.

One significant benefit of counting sort is that the time it takes only depends on n and k, it does not depend on the values of the data to be sorted. This could be a very useful property when designing a real-time system.

In conclusion, Counting Sort should only be used if k is low enough for it to be feasible and memory usage is not constrained, otherwise Insertion Sort is more practical.

References

[1] Paoloni, G. (2010). *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*. Intel Corporation.