Project 2- Sorting
CSCI 230 T Th 11:10 am
Compiler: g++
OS: Linux

Michael Morikawa

April 23, 2020

Notes

Status

All sorting algorithms work and give expected results

Extra Credit

Completed shell sort and radix sort(int keys)

Design Decisions

Main could have possibly been cleaner if I had come up with a better way of making sure to use the same unsorted list for each sorting algorithm, but I wanted all output to be generated with a single execution.

Data

Integer Key Results

Size:1,000

Algortihm	Key Comparisons	Data Moves	Time(microseconds)
Insertion	247,370	248,370	5498
QuickSort	11,479	7,107	313
QuickSort3	11,160	9,120	321
Merge	8,686	19,952	2,761
Radix	-	-	505
Shell	8,328	19,242	484

Size: 100,000

Algortihm	Key Comparisons	Data Moves	Time(microseconds)
Insertion	-	-	-
QuickSort	2,014,363	1,172,469	47,447
QuickSort3	1,937,431	1,360,674	50,065
Merge	1,536,302	3,337,856	401,306
Radix	-	-	65,222
Shell	3,032,564	4,966,856	120,764

String Key Results

Algortihm	Key Comparisons	Data Moves	Time(microseconds)
Insertion	244,350	245,355	7,513
QuickSort	11,245	7,092	410
QuickSort3	10,965	8,871	450
Merge	8,695	19,952	2,687
Radix	-	-	-
Shell	8,063	18,977	556

Size: 100,000

Algortihm	Key Comparisons	Data Moves	Time(microseconds)
Insertion	-	-	-
QuickSort	1,978,324	1,173,504	70,310
QuickSort3	1,919,654	1,365,048	72,456
Merge	1,536,328	3,337,856	426,894
Radix	-	-	-
Shell	2,779,124	4,713,416	151,346

Conclusions

From the data above we see that for each run QuickSort did the best, followed by the median of 3 QuickSort. There is some variation for the next fastest depending on the input size with radix sort being slightly slower than shell sort for 1k entries, but it is twice as fast for 100k entries. It being about the same for 1k might suggest that the constant factor associated with it is somewhat high. The next slowest is Merge sort which surpisingly is slower than shell sort. This might have to do how I implemented merge sort becuase I would have expected merge sort to be faster than shell. Insertion sort is last as expected.

While the key comparisons seem relatively the same for both string and integer, the entries with a string key performed slower than with an it, even with less comparisons and data moves. The reason is because when comparing two strings it will actually be multiple comparisons but I only keep track of the one that we see in code.

Source Code

main.cpp

```
#include <vector>
#include <iostream>
#include <string>
#include "Entry.hpp"
#include <fstream>
#include "Sorting.hpp"
#include "Comparators.hpp"

void loadFileIntKey(const char path[], std::vector<Entry<int, std::string>> &S);
void loadFileStringKey(const char path[], std::vector<Entry<std::string, int>> &S);

int main()
{
    typedef Entry<int, std::string> intEntry;
    typedef Entry<std::string, int> stringEntry;

    std::vector<intEntry> unsortedIntKeySmall;
    std::vector<stringEntry> unsortedStringKeySmall;
    std::vector<intEntry> unsortedIntKeyLarge;
    std::vector<stringEntry> unsortedStringKeyLarge;

    //Creating the different lists that need to be sorted

    loadFileIntKey("docs/large100k.txt", unsortedIntKeyLarge);
    loadFileIntKey("docs/small1k.txt", unsortedIntKeySmall);
    loadFileStringKey("docs/large100k.txt", unsortedStringKeyLarge);
    loadFileStringKey("docs/small1k.txt", unsortedStringKeySmall);

    //Copying the unsorted lists into different vectors to test different sorting algs
    //Probaly could have been done better.
    auto intInsertion = unsortedIntKeySmall;
    auto stringInsertion = unsortedStringKeySmall;
    auto intQuickMed3Large = unsortedIntKeyLarge;
    auto intQuickEndLarge = unsortedIntKeyLarge;
    auto stringQuickMed3Large = unsortedStringKeyLarge;
    auto stringQuickEndLarge = unsortedStringKeyLarge;
    auto intQuickMed3Small = unsortedIntKeySmall;
    auto intQuickEndSmall = unsortedIntKeySmall;
    auto stringQuickMed3Small = unsortedStringKeySmall;
    auto stringQuickEndSmall = unsortedStringKeySmall;
    auto intMergeSortLarge = unsortedIntKeyLarge;
    auto intMergeSortSmall = unsortedIntKeySmall;
    auto stringMergeSortLarge = unsortedStringKeyLarge;
    auto stringMergeSortSmall = unsortedStringKeySmall;
    auto intShellSmall = unsortedIntKeySmall;
    auto intShellLarge = unsortedIntKeyLarge;
    auto stringShellSmall = unsortedStringKeySmall;
    auto stringShellLarge = unsortedStringKeyLarge;
    auto intRadixSmall = unsortedIntKeySmall;
    auto intRadixLarge = unsortedIntKeyLarge;

    std::ofstream intOutput, stringOutput;
    intOutput.open("docs/intResults.txt");
    stringOutput.open("docs/stringResults.txt");

    Sorting<intEntry, EntryCompare<intEntry>> intSorter;
    Sorting<stringEntry, EntryCompare<stringEntry>> stringSorter;

    intOutput << "Integer Keys\n\n\n";

    intSorter.insertionSort(intInsertion, intOutput);
    intSorter.quickSortEnd(intQuickEndSmall, intOutput);
    intSorter.quickSortMedOf3(intQuickMed3Small, intOutput);
    intSorter.mergeSort(intMergeSortSmall, intOutput);
    intSorter.radixSort(intRadixSmall, intOutput);
    intSorter.shellSort(intShellSmall, intOutput);

    intSorter.quickSortEnd(intQuickEndLarge, intOutput);
    intSorter.quickSortMedOf3(intQuickMed3Large, intOutput);
    intSorter.mergeSort(intMergeSortLarge, intOutput);
    intSorter.radixSort(intRadixLarge, intOutput);
    intSorter.shellSort(intShellLarge, intOutput);

    stringOutput << "STRING KEYS\n\n\n";

    stringSorter.insertionSort(stringInsertion, stringOutput);
    stringSorter.quickSortEnd(stringQuickEndSmall, stringOutput);
    stringSorter.quickSortMedOf3(stringQuickMed3Small, stringOutput);
    stringSorter.mergeSort(stringMergeSortSmall, stringOutput);
    stringSorter.shellSort(stringShellSmall, stringOutput);
    stringSorter.quickSortEnd(stringQuickEndLarge, stringOutput);
    stringSorter.quickSortMedOf3(stringQuickMed3Large, stringOutput);
    stringSorter.mergeSort(stringMergeSortLarge, stringOutput);
    stringSorter.shellSort(stringShellLarge, stringOutput);
}

void loadFileIntKey(const char path[], std::vector<Entry<int, std::string>> &S)
{
    std::ifstream infile;
    infile.open(path);
    Entry<int, std::string> temp;
    int tempInt;
    if (infile.fail())
    {
        std::cout << "Couldn't find " << path;
        return;
    }
    while (true)
    {
        if (infile.eof())
        {
            break;
        }
        infile >> tempInt;
        temp.setKey(tempInt);
        temp.setValue(std::to_string(tempInt));
        S.push_back(temp);
    }
    infile.close();
}

void loadFileStringKey(const char path[], std::vector<Entry<std::string, int>> &S)
{
    std::ifstream infile;
    infile.open(path);
    Entry<std::string, int> temp;
    int tempInt;
    if (infile.fail())
    {
        std::cout << "Couldn't find " << path;
        return;
    }
    while (true)
    {
        if (infile.eof())
        {
            break;
        }
        infile >> tempInt;
        temp.setKey(std::to_string(tempInt));
        temp.setValue(tempInt);
        S.push_back(temp);
    }
    infile.close();
}
```

Sorting.hpp

```
#pragma once
#include <Entry.hpp>
#include <fstream>
#include <vector>
#include <chrono>

//TODO: Implement insertion sort
//TODO: Implement merge sort
//TODO: Track data moves for quick sort
//TODO: Utility output function

template <typename E, typename C>
class Sorting
{
public:
    void insertionSort(std::vector<E> &S, std::ofstream &outfile);
    void quickSortEnd(std::vector<E> &S, std::ofstream &outfile);
    void quickSortMedOf3(std::vector<E> &S, std::ofstream &outfile);
    void mergeSort(std::vector<E> &S, std::ofstream &outfile);

    //Extra Credit

    void shellSort(std::vector<E> &S, std::ofstream &outfile);
    void radixSort(std::vector<E> &S, std::ofstream &outfile);

protected:
    void
    quickSortStep(std::vector<E> &S, int a, int b, int pivotType,
                  int &compares, int &dataMoves);

    //puts the median of 3 at the end point, b, so we can use the same quicksort algorithm
    void medianOfThree(std::vector<E> &S, int a, int b, int &compares, int &dataMoves);

    //Helper function to output necessary info to separate txt doc
    void outputData(std::ofstream &outfile, const char *sortingType,
                    const std::vector<E> &S, int compares,
                    int dataMoves, float runtime);
    void mergeSortHelper(std::vector<E> &S, int &compares, int &dataMoves);

    void merge(std::vector<E> &S1, std::vector<E> &S2, std::vector<E> &S,
               int &compares, int &dataMoves);
    int getDigits(int n);
    int getMaxDigits(std::vector<E> &S);

private:
    C less;
};

template <typename E, typename C>
void Sorting<E, C>::insertionSort(std::vector<E> &S, std::ofstream &outfile)
{
    auto start = std::chrono::high_resolution_clock::now();
    E key;
    int j;
    int comparisons = 0;
    int dataMoves = 0;
    for (int i = 1; i < S.size(); i++)
    {
        key = S[i];
        dataMoves++;
        j = i - 1;
        while (j >= 0)
        {
            if (less(key, S[j]))
            {
                dataMoves++;
                comparisons++;
                S[j + 1] = S[j]; //swaps
                j--;
            }
            else
            {
                comparisons++;
                break;
            }
        }
        S[j + 1] = key;
        dataMoves++;
    }

    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    outputData(outfile, "Insertion Sort", S, comparisons, dataMoves, duration.count());
}

template <typename E, typename C>
void Sorting<E, C>::quickSortStep(std::vector<E> &S, int a, int b, int pivotType, int &compares, int &dataMove
{
    if (a >= b)
        return; // 0 or 1 left? done
    //If pivot type is 1 don't do anything if 2 move median to end
    if (pivotType == 2)
    {
        medianOfThree(S, a, b, compares, dataMoves);
    }

    E pivot = S[b];
    int l = a;     // left edge
    int r = b - 1; // right edge
    while (l <= r)
    {
        while (l <= r && !less(pivot, S[l]))
        {
            l++; // scan right till larger
            compares++;
        }
        while (r >= l && !less(S[r], pivot))
        {
            r--; // scan left till smaller
            compares++;
        }
        if (l < r) // both elements found
        {
            std::swap(S[l], S[r]);
            dataMoves += 3;
        }
    }
    std::swap(S[l], S[b]); // store pivot at l
    dataMoves += 3;
    quickSortStep(S, a, l - 1, pivotType, compares, dataMoves); // recur on both sides
    quickSortStep(S, l + 1, b, pivotType, compares, dataMoves);
}

template <typename E, typename C>
void Sorting<E, C>::medianOfThree(std::vector<E> &S, int a, int b, int &compares, int &dataMoves)
{
    int mid = (b + a) / 2;
    //Find and put median value at the end index
    if (less(S[mid], S[a]))
    {
        std::swap(S[a], S[mid]);
        dataMoves += 3;
    }
    if (less(S[b], S[a]))
    {
        std::swap(S[a], S[b]);
        dataMoves += 3;
    }
    if (less(S[mid], S[b]))
    {
        std::swap(S[mid], S[b]);
        dataMoves += 3;
    }
    compares += 3;
}

template <typename E, typename C>
void Sorting<E, C>::quickSortEnd(std::vector<E> &S, std::ofstream &outfile)
{
    int compares = 0;
    int dataMoves = 0;
    int pivotType = 1; //1 means use end as pivot
    if (S.size() <= 1)
    {
        return;
    }
    auto start = std::chrono::high_resolution_clock::now();
    quickSortStep(S, 0, S.size() - 1, pivotType, compares, dataMoves);
    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    outputData(outfile, "Quick Sort(End Pivot)", S, compares, dataMoves, duration.count());
}

template <typename E, typename C>
void Sorting<E, C>::quickSortMedOf3(std::vector<E> &S, std::ofstream &outfile)
{
    int compares = 0;
    int dataMoves = 0;
    int pivotType = 2; //2 means use median of 3 as pivot
    if (S.size() <= 1)
    {
        return;
    }
    auto start = std::chrono::high_resolution_clock::now();
    quickSortStep(S, 0, S.size() - 1, pivotType, compares, dataMoves);
    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    outputData(outfile, "Quick Sort(Median of 3)", S, compares, dataMoves, duration.count());
}

template <typename E, typename C>
void Sorting<E, C>::outputData(std::ofstream &outfile, const char *sortingType,
                               const std::vector<E> &S, int compares,
                               int dataMoves, float runtime)
{
    outfile << "============== " << sortingType << " ==============\n\n";
    outfile << "Size:" << S.size()
            << "\nNumber of compares: " << compares
            << "\nNumber of data moves:" << dataMoves
            << "\nRuntime: " << runtime << " microseconds"
            << "\nFirst 5 elements: \n";
    for (int i = 0; i < 5; i++)
    {
        outfile << S[i] << ' ';
    }
    outfile << "\n\nLast 5 elements: \n";
    for (int i = S.size() - 6; i < S.size(); i++)
    {
        outfile << S[i] << ' ';
    }
    outfile << "\n\n\n";
}

template <typename E, typename C>
void Sorting<E, C>::mergeSort(std::vector<E> &S, std::ofstream &outfile)
{

    int compares = 0;
    int dataMoves = 0;
    auto start = std::chrono::high_resolution_clock::now();
    mergeSortHelper(S, compares, dataMoves);
    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    outputData(outfile, "Merge Sort", S, compares, dataMoves, duration.count());
}
```

```
template <typename E, typename C>
void Sorting<E, C>::mergeSortHelper(std::vector<E> &S, int &compares, int &dataMoves)
{
    typedef typename std::vector<E>::iterator Iterator;
    int n = S.size();
    if (n <= 1)
    {
        return;
    }
    std::vector<E> S1, S2;
    Iterator p = S.begin();
    for (int i = 0; i < n / 2; i++)
    {
        S1.push_back(*p++);
        dataMoves++;
    }
    for (int i = n / 2; i < n; i++)
    {
        S2.push_back(*p++);
        dataMoves++;
    }
    S.clear();
    mergeSortHelper(S1, compares, dataMoves);
    mergeSortHelper(S2, compares, dataMoves);
    merge(S1, S2, S, compares, dataMoves);
}

template <typename E, typename C>
void Sorting<E, C>::merge(std::vector<E> &S1, std::vector<E> &S2, std::vector<E> &S, int &compares, int &dataM
{
    typedef typename std::vector<E>::iterator Iterator;
    Iterator p1 = S1.begin();
    Iterator p2 = S2.begin();

    while (p1 != S1.end() && p2 != S2.end())
    {
        if (less(*p1, *p2))
        {
            S.push_back(*p1++);
            dataMoves++;
        }
        else
        {
            S.push_back(*p2++);
            dataMoves++;
        }
        compares++;
    }

    while (p1 != S1.end())
    {
        S.push_back(*p1++);
        dataMoves++;
    }
    while (p2 != S2.end())
    {
        S.push_back(*p2++);
        dataMoves++;
    }
```

```
}

template <typename E, typename C>
void Sorting<E, C>::shellSort(std::vector<E> &S, std::ofstream &outfile)
{
    int interval = 0;
    int compares = 0;
    int dataMoves = 0;
    E key;
    int j;
    auto start = std::chrono::high_resolution_clock::now();
    while (interval < S.size() / 3)
    {
        interval = interval * 3 + 1;
    }
    (interval - 1) / 3;
    while (interval > 0)
    {
        for (int i = interval; i < S.size(); i++)
        {
            key = S[i];
            dataMoves++;
            j = i;
            while (j >= interval && less(key, S[j - interval]))
            {
                S[j] = S[j - interval];
                dataMoves++;
                compares++;
                j -= interval;
            }
            S[j] = key;
            dataMoves++;
        }
        interval = (interval - 1) / 3;
    }
    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    outputData(outfile, "Shell Sort", S, compares, dataMoves, duration.count());
}

template <typename E, typename C>
void Sorting<E, C>::radixSort(std::vector<E> &S, std::ofstream &outfile)
{
    std::vector<std::vector<E>> buckets(10);
    int bucketIndex, arrayIndex;
    int maxDigits = getMaxDigits(S);
    int pow10 = 1;
    auto start = std::chrono::high_resolution_clock::now();
    for (int digitIndex = 0; digitIndex < maxDigits; digitIndex++)
    {
        for (int i = 0; i < S.size(); i++)
        {
            bucketIndex = (S[i].key() / pow10) % 10;
            buckets[bucketIndex].push_back(S[i]);
        }
        arrayIndex = 0;
        for (int i = 0; i < 10; i++)
        {
            for (int j = 0; j < buckets[i].size(); j++)
            {
                S[arrayIndex++] = buckets[i][j];
            }
            buckets[i].clear();
        }
        pow10 *= 10;
    }
    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    outputData(outfile, "Radix Sort", S, -1, -1, duration.count());
}
template <typename E, typename C>
int Sorting<E, C>::getMaxDigits(std::vector<E> &S)
{
    int maxDigits = 0;
    int digitCount;
    for (int i = 0; i < S.size(); i++)
    {
        digitCount = getDigits(S[i].key());
        maxDigits = std::max(digitCount, maxDigits);
    }
    return maxDigits;
}

template <typename E, typename C>
int Sorting<E, C>::getDigits(int n)
{
    if (n == 0)
    {
        return 1;
    }
    int digits = 0;
    while (n != 0)
    {
        digits++;
        n /= 10;
    }
    return digits;
}
```

Entry.hpp

```
#pragma once
#include <iostream>
template <typename K, typename V>
class Entry
{
public:
    Entry(const K &k = K(), const V &v = V())
        : _key(k), _value(v) {}
    const K &key() const
    {
        return _key;
    }
    const V &value() const
    {
        return _value;
    }
    void setKey(const K &k)
    {
        _key = k;
    }
    void setValue(const V &v)
    {
        _value = v;
    }

    friend std::ostream &operator<<(std::ostream &output, const Entry E)
    {
        output << '(' << E._key << ',' << E._value << ')';
        return output;
    }

private:
    K _key;
    V _value;
};
```

Comparators.hpp

```
#pragma once
//Simple Comparator classes that just use "size" for comparison

template <typename T>
class HighToLow
{
public:
    bool operator()(const T &a, const T &b) const
    {
        return a > b;
    }
};

template <typename T>
class LowToHigh
{
public:
    bool operator()(const T &a, const T &b) const
    {
        return a < b;
    }
};

//Templated on an entry data type that supports the key() function
template <typename E>
class EntryCompare
{
public:
    bool operator()(const E &a, const E &b) const
    {
        return a.key() < b.key();
    }
};
```

Output

intResults.txt

```
Integer Keys


============== Insertion Sort ==============

Size:1000
Number of compares: 247370
Number of data moves:248370
Runtime: 5628 microseconds
First 5 elements:
(7,7) (11,11) (15,15) (39,39) (59,59)

Last 5 elements:
(8155,8155) (8163,8163) (8167,8167) (8175,8175) (8183,8183) (8191,8191)


============== Quick Sort(End Pivot) ==============

Size:1000
Number of compares: 11479
Number of data moves:7107
Runtime: 312 microseconds
First 5 elements:
(7,7) (11,11) (15,15) (39,39) (59,59)

Last 5 elements:
(8155,8155) (8163,8163) (8167,8167) (8175,8175) (8183,8183) (8191,8191)


============== Quick Sort(Median of 3) ==============

Size:1000
Number of compares: 11160
Number of data moves:9120
Runtime: 333 microseconds
First 5 elements:
(7,7) (11,11) (15,15) (39,39) (59,59)

Last 5 elements:
(8155,8155) (8163,8163) (8167,8167) (8175,8175) (8183,8183) (8191,8191)


============== Merge Sort ==============

Size:1000
Number of compares: 8686
Number of data moves:19952
Runtime: 5385 microseconds
First 5 elements:
(7,7) (11,11) (15,15) (39,39) (59,59)

Last 5 elements:
(8155,8155) (8163,8163) (8167,8167) (8175,8175) (8183,8183) (8191,8191)


============== Radix Sort ==============
Size:1000
Number of compares: -1
Number of data moves:-1
Runtime: 459 microseconds
First 5 elements:
(7,7) (11,11) (15,15) (39,39) (59,59)

Last 5 elements:
(8155,8155) (8163,8163) (8167,8167) (8175,8175) (8183,8183) (8191,8191)


=============== Shell Sort ===============

Size:1000
Number of compares: 8328
Number of data moves:19242
Runtime: 484 microseconds
First 5 elements:
(7,7) (11,11) (15,15) (39,39) (59,59)

Last 5 elements:
(8155,8155) (8163,8163) (8167,8167) (8175,8175) (8183,8183) (8191,8191)


=============== Quick Sort(End Pivot) ===============

Size:100000
Number of compares: 2014363
Number of data moves:1172469
Runtime: 54450 microseconds
First 5 elements:
(1,1) (2,2) (3,3) (4,4) (5,5)

Last 5 elements:
(99995,99995) (99996,99996) (99997,99997) (99998,99998) (99999,99999) (100000,100000)


=============== Quick Sort(Median of 3) ===============

Size:100000
Number of compares: 1937431
Number of data moves:1360674
Runtime: 53857 microseconds
First 5 elements:
(1,1) (2,2) (3,3) (4,4) (5,5)

Last 5 elements:
(99995,99995) (99996,99996) (99997,99997) (99998,99998) (99999,99999) (100000,100000)


=============== Merge Sort ===============

Size:100000
Number of compares: 1536302
Number of data moves:3337856
Runtime: 436272 microseconds
First 5 elements:
(1,1) (2,2) (3,3) (4,4) (5,5)
```

```
Last 5 elements:
(99995,99995) (99996,99996) (99997,99997) (99998,99998) (99999,99999) (100000,100000)


============== Radix Sort ==============

Size:100000
Number of compares: -1
Number of data moves:-1
Runtime: 68513 microseconds
First 5 elements:
(1,1) (2,2) (3,3) (4,4) (5,5)

Last 5 elements:
(99995,99995) (99996,99996) (99997,99997) (99998,99998) (99999,99999) (100000,100000)


============== Shell Sort ==============

Size:100000
Number of compares: 3032564
Number of data moves:4966856
Runtime: 120764 microseconds
First 5 elements:
(1,1) (2,2) (3,3) (4,4) (5,5)

Last 5 elements:
(99995,99995) (99996,99996) (99997,99997) (99998,99998) (99999,99999) (100000,100000)
```

---

stringResults.txt

```
STRING KEYS


============== Insertion Sort ==============

Size:1000
Number of compares: 244350
Number of data moves:245355
Runtime: 7968 microseconds
First 5 elements:
(103,103) (1035,1035) (1047,1047) (1055,1055) (1063,1063)

Last 5 elements:
(95,95) (955,955) (959,959) (987,987) (99,99) (995,995)


============== Quick Sort(End Pivot) ==============

Size:1000
Number of compares: 11245
Number of data moves:7092
Runtime: 410 microseconds
First 5 elements:
(103,103) (1035,1035) (1047,1047) (1055,1055) (1063,1063)

Last 5 elements:
(95,95) (955,955) (959,959) (987,987) (99,99) (995,995)
```

```
=============== Quick Sort(Median of 3) ===============

Size:1000
Number of compares: 10965
Number of data moves:8871
Runtime: 440 microseconds
First 5 elements:
(103,103) (1035,1035) (1047,1047) (1055,1055) (1063,1063)

Last 5 elements:
(95,95) (955,955) (959,959) (987,987) (99,99) (995,995)


=============== Merge Sort ===============

Size:1000
Number of compares: 8695
Number of data moves:19952
Runtime: 2601 microseconds
First 5 elements:
(103,103) (1035,1035) (1047,1047) (1055,1055) (1063,1063)

Last 5 elements:
(95,95) (955,955) (959,959) (987,987) (99,99) (995,995)


=============== Shell Sort ===============

Size:1000
Number of compares: 8063
Number of data moves:18977
Runtime: 556 microseconds
First 5 elements:
(103,103) (1035,1035) (1047,1047) (1055,1055) (1063,1063)

Last 5 elements:
(95,95) (955,955) (959,959) (987,987) (99,99) (995,995)


=============== Quick Sort(End Pivot) ===============

Size:100000
Number of compares: 1978324
Number of data moves:1173504
Runtime: 70310 microseconds
First 5 elements:
(1,1) (10,10) (100,100) (1000,1000) (10000,10000)

Last 5 elements:
(99994,99994) (99995,99995) (99996,99996) (99997,99997) (99998,99998) (99999,99999)


=============== Quick Sort(Median of 3) ===============

Size:100000
Number of compares: 1919654
Number of data moves:1365048
Runtime: 72456 microseconds
First 5 elements:
(1,1) (10,10) (100,100) (1000,1000) (10000,10000)
```

```
Last 5 elements:
(99994,99994) (99995,99995) (99996,99996) (99997,99997) (99998,99998) (99999,99999)


============== Merge Sort ==============

Size:100000
Number of compares: 1536328
Number of data moves:3337856
Runtime: 426894 microseconds
First 5 elements:
(1,1) (10,10) (100,100) (1000,1000) (10000,10000)

Last 5 elements:
(99994,99994) (99995,99995) (99996,99996) (99997,99997) (99998,99998) (99999,99999)


============== Shell Sort ==============

Size:100000
Number of compares: 2779124
Number of data moves:4713416
Runtime: 151346 microseconds
First 5 elements:
(1,1) (10,10) (100,100) (1000,1000) (10000,10000)

Last 5 elements:
(99994,99994) (99995,99995) (99996,99996) (99997,99997) (99998,99998) (99999,99999)
```