Lab 3 Sorting Pt. 1

Michael Morikawa

March 12, 2020

Lab Questions

Question 1 What are some good reasons for sorting a list of values?
You want to sort a list of values if you are going to be searching for those values often so that you can take advantage of binary search algorithms.

Question 2 What are the three steps in a divide-and-conquer algorithm design pattern?
First step is to divide which means split the input data into different subsets to work with a smaller amount of data. Then you recur which is recursively solving the problem with the subsets. Then you conquer which combines the solutions to the problems from the subsets.

Source Code

insertionSort.cpp

```
#include <chrono>
#include <iostream>

//Returns pair corresponding to Comparisons and Swaps
std::pair<int, int> insertionSort(int arr[], int size);

void randomizeArr(int arr[], int size);

//Extra Credit, Create and measures runtime for insertion sort
//on an array of given size
void measureRuntime(int size);

int main()
{
    int sortedArr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int descendingArr[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
    int randArr[10];
    std::pair<int, int> data;
    randomizeArr(randArr, 10);

    std::cout << "Sorted Array: \n";
    data = insertionSort(sortedArr, 10);
    std::cout << "Array sorted with " << data.first << " comparisons and "
              << data.second << " swaps\n\n";

    std::cout << "Reverse Sorted Array: \n";
    data = insertionSort(descendingArr, 10);
```

```
    std::cout << "Array sorted with " << data.first << " comparisons and "
              << data.second << " swaps\n\n";

    std::cout << "Random Array: \n";
    data = insertionSort(randArr, 10);
    std::cout << "Array sorted with " << data.first << " comparisons and "
              << data.second << " swaps\n\n";

    //Extra Credit

    std::cout << "Collecting Insertion Sort runtimes:\n\n";

    measureRuntime(1000);
    measureRuntime(10000);
    measureRuntime(100000);
}

std::pair<int, int> insertionSort(int arr[], int size)
{
    int key;
    int j;
    int comparisons = 0;
    int swaps = 0;
    for (int i = 1; i < size; i++)
    {
        key = arr[i];
        j = i - 1;
        while (j >= 0)
        {
            if (arr[j] > key)
            {
                swaps++;
                comparisons++;
                arr[j + 1] = arr[j]; //swaps
                j--;
            }
            else
            {
                comparisons++;
                break;
            }
        }
        arr[j + 1] = key;
    }
    std::pair<int, int> data(comparisons, swaps);
    return data;
}

void randomizeArr(int arr[], int size)
{
    std::srand(time(NULL));
    for (int i = 0; i < size; i++)
    {
        arr[i] = rand() % 100 + 1;
```

```
    }
}

void measureRuntime(int size)
{
    int* arr = new int[size];
    randomizeArr(arr, size);
    auto start = std::chrono::high_resolution_clock::now();
    insertionSort(arr, size);
    auto end = std::chrono::high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);

    std::cout << "Array of size " << size << " sorted in " << duration.count()
              << " microseconds or " << duration.count() / (float)1000000 << " seconds\n";

    delete[] arr;
}
```

mergeSort.cpp

```
#include <chrono>
#include <iostream>
#include <list>

template <typename E>
class LessThan
{
public:
    bool operator()(const E& first, const E& second) const
    {
        return first < second;
    }
};

template <typename E, typename C>
void mergeSort(std::list<E>& S, const C& less);

template <typename E, typename C>
void merge(std::list<E>& S1, std::list<E>& S2, std::list<E>& S, const C& less);

std::list<int> randomizeList(int size);

void measureRuntime(int size);

template <typename E>
void printSeq(const E& seq);

int main()
{
    std::list<int> randList = randomizeList(10);
    LessThan<int> less;

    std::cout << "Unsorted List:\n";
```

```
    printSeq(randList);

    mergeSort(randList, less);
    std::cout << "\nSorted with Merge Sort:\n";
    printSeq(randList);

    //Extra credit

    std::cout << "\n\nCollecting Merge Sort Runtimes:\n\n";

    std::list<int> testList1 = randomizeList(1000);
    std::list<int> testList2 = randomizeList(10000);
    std::list<int> testList3 = randomizeList(100000);

    measureRuntime(1000);
    measureRuntime(10000);
    measureRuntime(100000);
}

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

template <typename E, typename C>
void merge(std::list<E>& S1, std::list<E>& S2, std::list<E>& S, const C& less)
{
    typedef typename std::list<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++);
        }
        else
        {
            S.push_back(*p2++);
        }
    }

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

template <typename E>
void printSeq(const E& seq)
{
    for (auto i : seq)
    {
        std::cout << i << '\n';
    }
}

std::list<int> randomizeList(int size)
{
    std::srand(time(NULL));
    std::list<int> randList;
    for (int i = 0; i < size; i++)
    {
        randList.push_back(std::rand() % 1000);
    }

    return randList;
}

void measureRuntime(int size)
{
    std::list<int> testList = randomizeList(size);
    LessThan<int> less;
    auto start = std::chrono::high_resolution_clock::now();
    mergeSort(testList, less);
    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    std::cout << "List of size " << size << " sorted in " << duration.count() << " microseconds or "
              << duration.count() / (float)1000000 << " seconds\n";
}
```

Output

Insertion Sort

```
Sorted Array:
Array sorted with 9 comparisons and 0 swaps

Reverse Sorted Array:
Array sorted with 45 comparisons and 45 swaps

Random Array:
Array sorted with 31 comparisons and 24 swaps
```

Merge Sort

```
Unsorted List:
645
560
660
270
312
140
789
404
576
471

Sorted with Merge Sort:
140
270
312
404
471
560
576
645
660
789
```

Extra Credit

```
Collecting Insertion Sort runtimes:

Array of size 1000 sorted in 974 microseconds or 0.000974 seconds
Array of size 10000 sorted in 79723 microseconds or 0.079723 seconds
Array of size 100000 sorted in 6513293 microseconds or 6.51329 seconds
```

```
Collecting Merge Sort Runtimes:

List of size 1000 sorted in 4888 microseconds or 0.004888 seconds
List of size 10000 sorted in 58970 microseconds or 0.05897 seconds
List of size 100000 sorted in 539641 microseconds or 0.539641 seconds
```