Programming Lab 2

Insertion Sort

My holdover sorting algorithm from Lab 1 is the Insertion Sort. I built my `LinkedList` class such that all insertions are sorted into ascending order at the time of insertion. This is not very efficient. It's fine for small values, but will grow at a dramatic rate as the number of items increases.

```
void LinkedList::insert(Node* newNode) {
    //inserts nodes in ascending order, sorted by data value
    if (Head == nullptr) {//Just add it if the list is empty
        Head = newNode;
    }
    else if (*(Head) > *(newNode)) {
        //if smaller than the current head of list, insert there
        newNode->setNext(Head);
        Head = newNode;
    }
    else if (Head->getNext() == nullptr) {
        //add to the end of a single item list
        Head->setNext(newNode);
    }
    else {
        Node* next = Head;
        while (*(next->getNext()) < *(newNode)) {
            //iterate until next value is null or smaller than new value
            next = next->getNext();
            if (next->getNext() == nullptr) {
                next->setNext(newNode);
                return;
            }
        }
        newNode->setNext(next->getNext());
        next->setNext(newNode);
    }
}
```

Given n items to insert, each insertion increases the size of the list by one, to a maximum size of n. In the best possible case, the list is already sorted prior to insertion, and will complete in n iterations, just pushing the item in at the list head each time. In the worst case, items are given to the list in inverse order, and every insertion must traverse all the way to the end of the list. On average, the sorting will fall somewhere in between; call it half of the current list size for simplicity.

Given this set of parameters, we can find that the sort will run in the following worst, best, and average time complexities:

- $O(\sum_{i=0}^{n} n(n-i)+5i) \sim O(n^2)$
- $\Omega(5n) \sim \Omega(n)$
- $\Theta(\sum_{i=0}^{n} \frac{n(n-i)}{2}) \sim O(n^2)$

This puts insertion sort in the awkward category of being among the worse sorting algorithms in the worst and average cases, growing exponentially with n. And while it is capable of being among the fastest, this only happens with data *that has already been sorted*, which negates whatever benefit there might be.

Radix Sort

Radix sort is a relatively fast algorithm. It breaks the problem of sorting integers into as many pieces as the length of the longest integer string in the dataset to be sorted. Unlike merge sort, which breaks the problem in half, and half again until down to one and two pieces, requiring $nlog_2(n)$ iterations, the Radix Sort can move much faster by breaking it into $log_{10}(max(n))$ pieces, with a maximum (sorting standard unsigned integers) of $10n$ iterations.

```
void radixSort(std::vector<int>& vector) {
    /*Sorts a vector of integers by doing a count sort on each
    place(1, 10, etc) for each significant figure in the largest integer*/
    int max = maxInt(vector);
    for (int place = 1; max / place > 0; place *= 10) {
        vector = countSort(vector, place);
    }
```

Radix sort does this by using a Counting Sort subroutine to sort the dataset by each position in the string, sorting by a larger power of 10 each time until the numbers have been sorted by all available places. See my writeup of Counting Sort below for that algorithmic analysis, but suffice to say that the counting sort has time complexity of $O(n)$, and is run no more than 10 times to sort a dataset containing only unsigned standard-length integers.

- $O(n \times log_{10}(MAX\_INT)) \approx O(n \times 10) \sim O(n)$
- $\Omega(n \times log_{10}(9)) \approx \Omega(n)$
- $\Theta(n \times log_{10}(log_2(2147483647))) \approx \Theta(n \times 5) \sim \Theta(n)$

Radix Sort has limitations; it can only be used for integers or data that can be represented as integers, and is best used for relatively short integers. If the number of places in each integer is equal to or greater than n, the time can actually approach $O(n^2)$. It also does not sort in place, the way that some other sorting algorithms do; the items are copied from one data structure into another in sorted order, using more system memory in the process.

Counting Sort

Counting Sort is what I used as a subroutine to accomplish my Radix Sort. Counting Sort is a non-comparative sort. Where Insertion Sort and many others compare elements and swap them, there is no need to do that in a counting sort. A hashtable is made of the occurrences of the possible values of data to sort. Each count is added to the next to allow the algorithm to mathematically place each item in the correct range of indices when populating the sorted data structure.

```
std::vector<int> countSort(std::vector<int>& vector, int place)
{
    /*Sorts a vector of integers into ascending order by the value in a given
place(1, 10, etc)
    Uses a hashtable of value counts and an offset by preceding values to
determine indicies of         placement integers with matching significant
figures will retain original order*/
    std::vector<int> output;
```

```
    output.resize(vector.size());
    int count[10] = { 0 };

    //uses count as a hashtable of counts of which digit(1, 2, ..., 9) is in the
nth place(1, 10, etc)
    for (int i = 0; i < vector.size(); i++) {
        count[(vector.at(i) / place) % 10]++;
    }
    //each set will be inserted starting at the position dictated by counts of
    //all numbers with smaller first digits
    //the count vector is reused for this
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }
    //using the counts in conjunction with the correct place digit, copy the
values from vector
    //into the correct order in output
    for (int i = vector.size() - 1; i >= 0; i--){
        output.at(count[(vector.at(i) / place) % 10] - 1) = vector.at(i);
        count[(vector.at(i) / place) % 10]--;
    }
    return output;
}
```

Rather than comparing n items to each other n times, it can simply iterate and count each item's value into the hashtable. Then it iterates across the possible values, 10 in its use for a radix sort, combining the numbers so that they can be placed in the range they ought to. And finally another iteration across the data structure to be sorted, in reverse for ascending order, taking each data value and copying it into the last index of the correct range for items with that value.

The time complexity for each individual Counting Sort as part of a Radix sort is:

- $O(2n + 10) \sim O(n)$
- $\Omega(2n + 10) \sim \Omega(n)$
- $\Theta(2n + 10) \sim \Theta(n)$

Like Radix Sort, counting sort can be of limited utility. There has to be a relatively small, fixed number of possible values with a clearly defined comparison; it works as part of Radix Sort because there are only 10 possible values at each iteration when dealing with base 10 numbers, and they have a clearly defined sequence to sort by. It also does not sort in place, and requires additional memory to hold the sorted data at the same time as the unsorted data.