Week 5 Assignment

CSE 503 – Summer 2024

Richard Douglas

Assignment Description

We were asked to ecomplete the following exercises regarding sorting algorithms:

7.1 Sort the sequence 3, 1, 4, 1, 5, 9, 2, 6, 5 using insertion sort. Show each iteration.
7.2 What is the running time of insertion sort if all elements are equal?
7.15 Sort 3, 1, 4, 1, 5, 9, 2, 6 using mergesort. Show each iteration.
7.19 Sort 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 using quicksort with median-of-three partitioning and a cutoff of 3. Show each iteration.

Logic Employed

Oh boy.... there are really three sets of logic employed in this coding assignment, because we implemented three different sorting algorithms. I will explain these each in turn: **Insertion Sort, Merge sort, Quick Sort.**

Insertion Sort

In this algorithm, each element of an array is compared with each other to find its appropriate index. The subarrray up to the element being evaluated is considered sorted. The element being evaluated is compared to each of the elements in the sorted array. Once the value of the element being evaluated is less than the element it is being compared to, it is inserted into the index after the element it was just assessed to be of greater value than. This nested loop –a loop over all lower idx elements, for all elements greater than 0 – creates a quadratic O(n^2) time complexity for very unsorted arrays. However, little time is used if the array is already nearly sorted. In fact, as we observed in sorting our array where all elements are the same, O(n) time complexity is achieved. Fortunately, there is little extra space used – only the extra space for the temporary pointer and two iterator objects being compared, so the space complexity is straight O(1).

The code I implemented is below

```
template <typename Comparable, typename Iterator, typename Comparator>
void insertionSortHelp(vector<Comparable> &a, Iterator begin, Iterator end, Comparator
lessThan)
{
Iterator j;
//starting at idx 1 (second element), loop to the end.
for (Iterator p = begin + 1; p != end; ++p)
{
printMyList(a);
auto tmp = *p;
// loop through sorted array
for (j = p; j != begin && lessThan(tmp, *(j - 1)); --j)
{
```

```
*j = *(j - 1);
}
//assign tmp pointer to j pointer, inserting the Iterator object
*j = tmp;
}
}
```

Merge Sort

Merge sort is just as simple to wrap your head around as insertion sort. The process is recursive, and this algorithm is a divide and conquer algorithm. Put simply, the array is split into subarrays until each subarray contains only one element. All subarrays are compared, then merged in such a way that the merged array is in sorted order.

The merge algorithm is the toughest bit to understand. Basically, all of the elements of each sorted subarray are compared and copied to a new array. First, we initialize pointers to the first element of each subarray being compared. The smaller element is copied to the new, merged array. Then the pointer for the element that was copied is incremented, and the next element from the subarray is compared with the same element from the other subarray. This process is repeated until one of the pointers reaches the end of its subarray. Then, the remaining elements from the other subarray can be copied to the merged array.

The time complexity for this algorithm is O(N * logN) the recursive nature of solving this problem reduces the number of comparisons that must be made from insertion sort. However, we still must compare all elements of each subarray. What's really great about merge sort is that it has this time complexity in all cases!

My code, including the code for the merge process is below:

```
//merge routine
void merge(vector<int> & a, vector<int> & temp, int left_p, int right_p, int right_end)
{
int left_end = right_p - 1;
int temp_p = left_p;
int num_elems = right_end - left_p + 1;

while (left_p <= left_end && right_p <= right_end)
{
if (a[left_p] <= a[right_p])
temp[temp_p++] = a[left_p++];
else
temp[temp_p++] = a[right_p++];
}
while (left_p <= left_end)
temp[temp_p++] = a[left_p++];
while (right_p <= right_end)
temp[temp_p++] = a[right_p++];
// printMyList(a);
// printMyList(temp);
for (int i = 0; i<num_elems; i++, right_end--)
a[right_end] = temp[right_end];
}
// merge sort logic
void merge_sort(vector<int> & a, vector<int> & temp, int left, int right)
{
printMyList(a);
if (left < right)
{
int center = (left+right)/2;
merge_sort(a, temp, left, center);
merge_sort(a, temp, center+1, right);
merge(a, temp, left, center+1, right);
}
// printMyList(a);
}
// driver function
void merge_sort ( vector<int> & a )
{
vector<int> temp(a.size());
merge_sort(a, temp, 0, a.size()-1);
// printMyList(a);
}
```

Quick sort

Like merge, quick sort uses a divide and conquer approach. However, instead of dividing an array equally by default, a pivot point is chosen. This can be selected in a variety of ways, including pseudo random number generation within a range... However, when the pivot point is always the first or last element in the array or subarrays, a worst-case time complexity of O(N^2) is achieved. This is obviously less than ideal. We used a "median of three" strategy in which we use the element with the median value between the first, middle and last element of an array, which tends to avoid such a bad case.

Once we pick our pivot, we divide the array into two subarrays – one containing all elements less than the partition and one containing all elements greater than the partition. This logic is called recursively on all subarrays down to some lower limit (cutoff) subarray size – three in our case. A merge routine is not required. We are sorting "in place" here.

The best and average case time complexities are O(N log N). One might wonder "why would I use this algorithm when merge sort has the same time complexity for ALL cases, without such a horrible worst case???" but in practice, quick sort does not often have a worst case run time, and there are measures one can take to reduce the likelihood of such poor performance. Additionally, since sorting occurs in-place, no merge routine is required, which speeds up the runtime.

My code is provided below:

```
int median3(vector<int> &a, int left, int right)
{
int center = (left + right) / 2;
if(a[center] < a[left])
swap(a[left], a[center]);
if(a[right] < a[left])
swap(a[left], a[right]);
if(a[right] < a[center])
swap(a[center], a[right]);
swap(a[center], a[right-1]);
return a[right-1];
}

// quick sort routine
void quick_sort(vector<int> & a, int left, int right)
{
printMyList(a);
if (left+3 <= right)
{
```

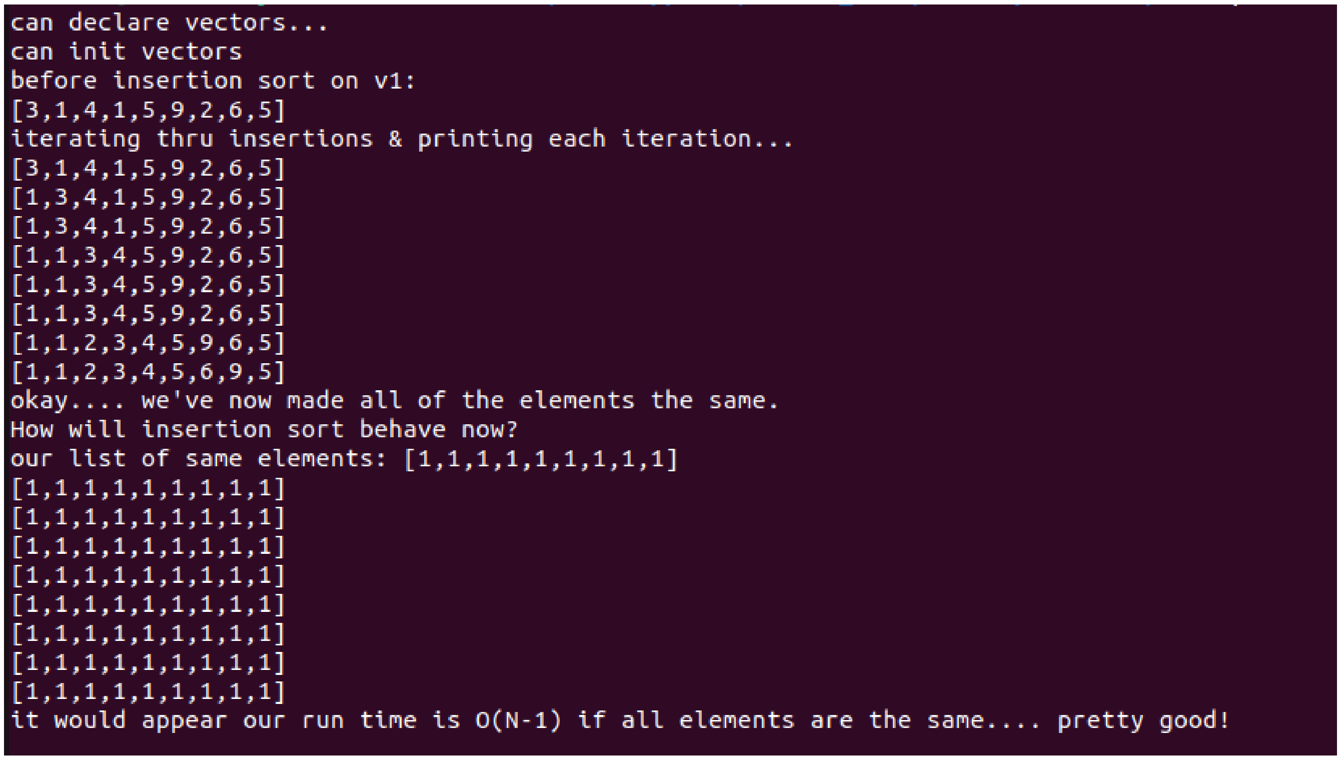
```
int pivot = median3(a, left, right);
int i, j;
i = left;
j = right-1;
for( ; ; )
{
while(a[++i]<pivot){}
while(a[--j]>pivot){}
if (i < j)
swap(a[i], a[j]);
else
break;
}
swap(a[i], a[right-1]);
quick_sort(a, left, i-1);
quick_sort(a, i+1, right);
}
else
insertionSort(a, a.begin()+left, a.begin()+ right+1, less<int>());
}

void quick_sort(vector<int> & a)
{
quick_sort(a, 0, a.size()-1);
}
```

I/O

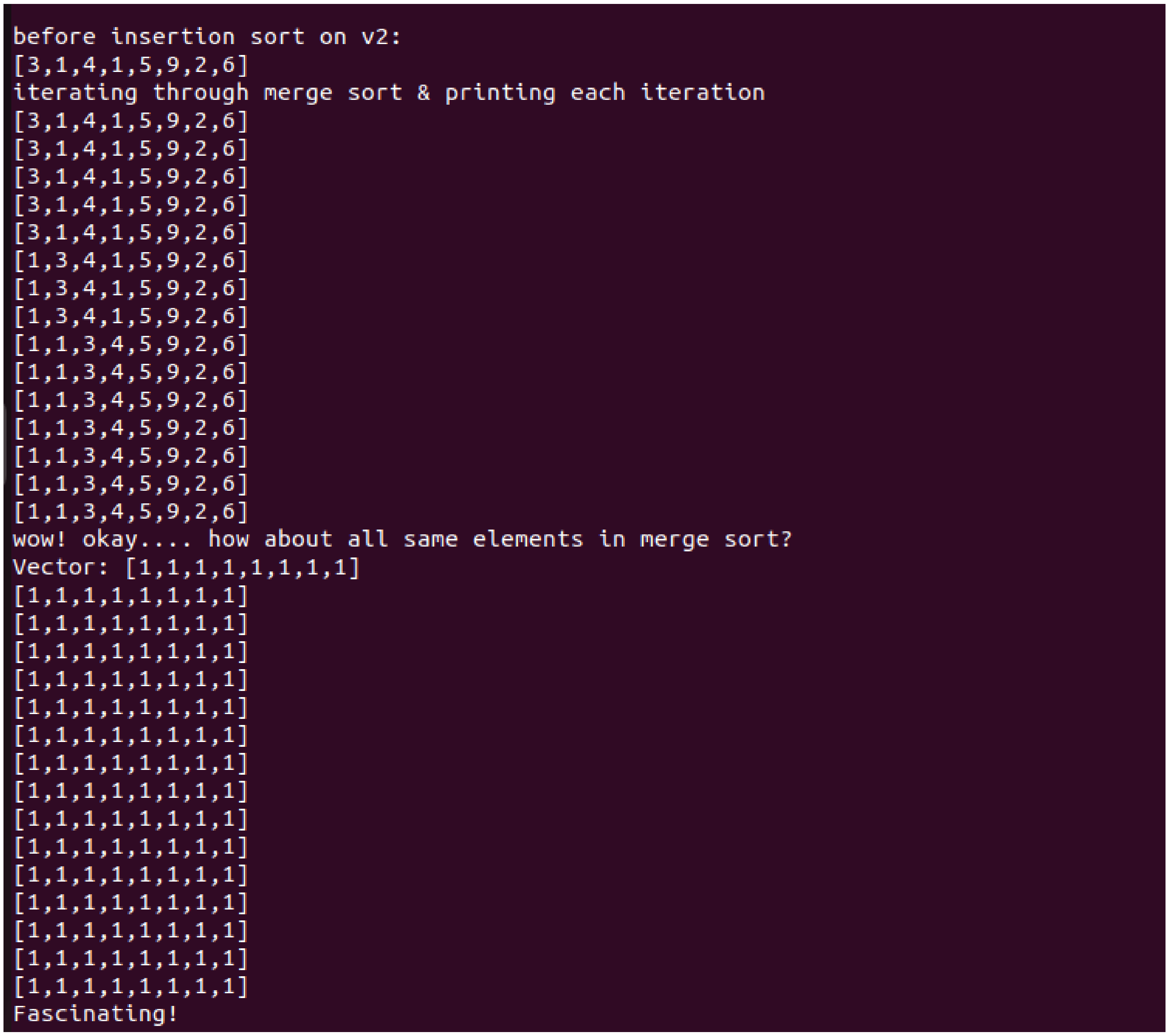
The user is not prompted for input. Instead we were asked to initialize vectors (or some other inerrable object like a list). These objects are then passed into our sorting algorithms.

Screenshots

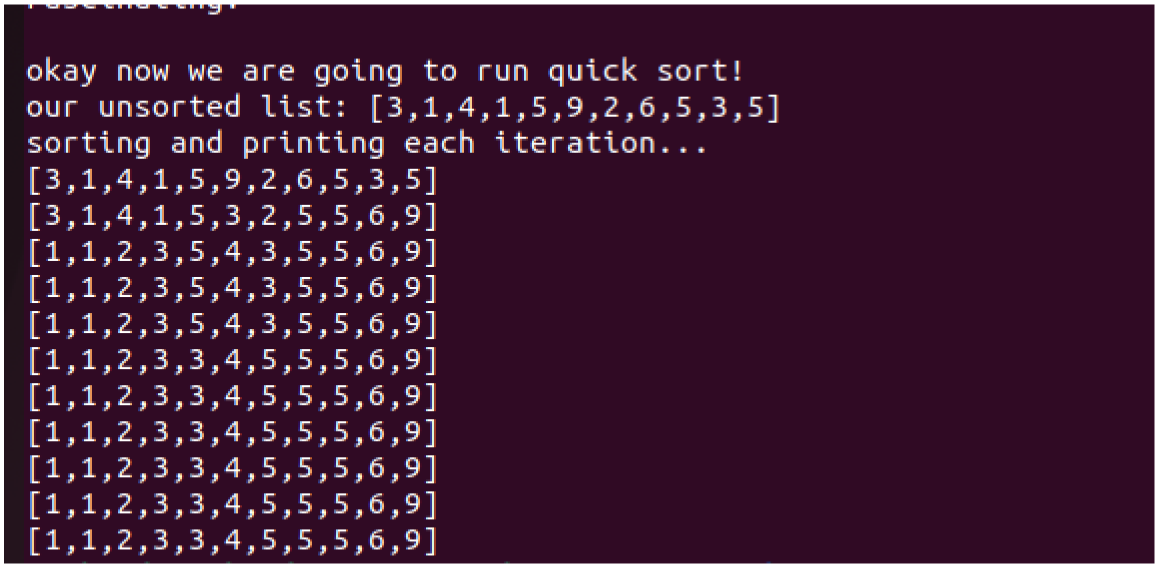
The following is the output run in my terminal.



```
can declare vectors...
can init vectors
before insertion sort on v1:
[3,1,4,1,5,9,2,6,5]
iterating thru insertions & printing each iteration...
[3,1,4,1,5,9,2,6,5]
[1,3,4,1,5,9,2,6,5]
[1,3,4,1,5,9,2,6,5]
[1,1,3,4,5,9,2,6,5]
[1,1,3,4,5,9,2,6,5]
[1,1,3,4,5,9,2,6,5]
[1,1,2,3,4,5,9,6,5]
[1,1,2,3,4,5,6,9,5]
okay.... we've now made all of the elements the same.
How will insertion sort behave now?
our list of same elements: [1,1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1,1]
it would appear our run time is O(N-1) if all elements are the same.... pretty good!
```



```
before insertion sort on v2:
[3,1,4,1,5,9,2,6]
iterating through merge sort & printing each iteration
[3,1,4,1,5,9,2,6]
[3,1,4,1,5,9,2,6]
[3,1,4,1,5,9,2,6]
[3,1,4,1,5,9,2,6]
[3,1,4,1,5,9,2,6]
[1,3,4,1,5,9,2,6]
[1,3,4,1,5,9,2,6]
[1,3,4,1,5,9,2,6]
[1,1,3,4,5,9,2,6]
[1,1,3,4,5,9,2,6]
[1,1,3,4,5,9,2,6]
[1,1,3,4,5,9,2,6]
[1,1,3,4,5,9,2,6]
[1,1,3,4,5,9,2,6]
[1,1,3,4,5,9,2,6]
wow! okay.... how about all same elements in merge sort?
Vector: [1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
[1,1,1,1,1,1,1,1]
Fascinating!
```



```
okay now we are going to run quick sort!
our unsorted list: [3,1,4,1,5,9,2,6,5,3,5]
sorting and printing each iteration...
[3,1,4,1,5,9,2,6,5,3,5]
[3,1,4,1,5,3,2,5,5,6,9]
[1,1,2,3,5,4,3,5,5,6,9]
[1,1,2,3,5,4,3,5,5,6,9]
[1,1,2,3,5,4,3,5,5,6,9]
[1,1,2,3,3,4,5,5,5,6,9]
[1,1,2,3,3,4,5,5,5,6,9]
[1,1,2,3,3,4,5,5,5,6,9]
[1,1,2,3,3,4,5,5,5,6,9]
[1,1,2,3,3,4,5,5,5,6,9]
[1,1,2,3,3,4,5,5,5,6,9]
```

Conclusions

This exercise was more difficult than previous ones. The textbook does not implement insertion sort in a way that makes immediate sense to be called in quick sort. This caused me a fair amount of consternation.