COMP26120 Lab 5

Oliver O'Hara

December 9, 2019

1 Complexity Analysis

1.1 Insertion Sort

Insertion sort works using a relatively primitive idea of sorting, taking each value of the data set in turn and placing it in the correct position. Despite being relatively simple to understand and implement, it isn't particularly efficient which becomes more noticeable as the amount of data to be sorted increases.

```
void insertion_sort(struct darray* arr)
{
   //Loop through all the elements in the array, inserting them into the correct
   //position
   for(int index = 1; index < arr->size; index++)
   {
      //Pick out the value to be inserted
      Value_Type insert_value = arr->cells[i];
      int j = i - 1;

      //While the value to the left of the one to be inserted is less than it
      //swap the values around
      while(j >= 0 && (compare(arr->cells[j], insert_value)) > 0)
      {
         arr->cells[j + 1] = arr->cells[j];
         j = j - 1;
      } //while
      //Move the value back into the correct place once it is found
      arr->cells[j + 1] = insert_value;
   } //for
} //insertion_sort
```

Assumed complexity:$O(n^2)$

Proof:
As can be seen from the code above, the for loop will run through all the elements in the array with the while loop inside it running at most j - 1 times. The rest of the lines in the code are independent of the size of the array and so are viewed as constants...

$$\sum_{j=2}^{n} c1 + (j-1)c2 + c3$$

Leaving us with...

$$d_1 n^2 + d_2 n + d_3$$

And so we can see that the complexity is given by $O(n^2)$

Best Case:
The best case scenario occurs when the data set is already sorted and so the inner while loop is never triggered, meaning that only the outer for loop will run. It is clear to see with the proof above that this will give us a best case complexity of $O(n)$.

Worst Case:
The worst case occurs when the data is sorted, but this time in reverse order. This would cause the inner while loop to run for every value in the data set. As the proof above shows, in this situation the complexity of the algorithm is given by $O(n^2)$.

Average Case:
For a randomly sorted array the proof above still holds, with a high probability that the inner while loop is triggered at some point in the execution of the algorithm. Again this gives us a complexity of $O(n^2)$.

1.2 Quick Sort

Quick sort is a more involved sorting algorithm that involves selecting an item in the data set to act as a pivot, with any items that are smaller than it being moved to the left and any larger items being moved to the right. This process repeats with a different pivot until all the items in the set are sorted. This algorithm makes use of the divide and conquer approach, meaning it is often more efficient than insertion sort.

```
//Helper function that takes the last element as a pivot, places it at the
//correct position. All the smaller items are placed on the left, all the larger
//ones on the right
```

```
int partition(struct darrays* arr, int low, int high)
{
   //Pivot value
   Value_Type pivot = arr->cells[high];
   int i = low -1

   // If the item pointed to by j is less than the pivot than increment the
   //index of the smaller element pointer to by i.
   for(int j = low; j <= high - 1; j++)
   {
      if(compare(arr->cells[j], pivot) < 0)
      {
         i++;
         swap(&arr->cells[i], &arr->cells[j]);
      } //if
   } //for
   swap(&arr->cells[i + 1], &arr->cells[high]);
   return(i + 1);
} //partition

void quick_sort(struct darray* arr, int low, int high)
{
   //As long as the starting index is less than the ending index
   if(low < high)
   {
      //Find the partition index using the helper function
      int partition_index = partition(arr, low, high);
      //Recursively call to sort the elements before and after the partition
      quick_sort(arr, low, (partition_index - 1));
      quick_sort(arr, (partition_index + 1), high);
   } //if
} //quick_sort
```

Assumed complexity: $O(nlog(n))$

Proof:
The helper method shown above the quick sort function ensures that this implementation will always run with the best case scenario. This occurs when the partitions are relatively well balanced, with the elements to the left being smaller the value of the pivot and the larger values being found to the right of the pivot. The complexity of this algorithm is $O(n)$ which doesn't require formal proof as it simply runs through the data using a for loop. Using this helper function we can assume that the the average partition will contain n/2 or n-1/2 (depending on whether the original data set is even or odd) which will be split again in half for an average case. The continuous splitting of the data

set in this recursive manner (along with the helper function) gives us an overall average case complexity of $O(nlog(n))$.

Best Case:
The best case scenario occurs when the partitions are of roughly equal size as described above, with the proof above applying giving us a complexity of$O(nlog(n))$.

Worst Case:
The worst case scenario for this algorithm occurs when the partitions created are as unbalanced as possible (for example having only one element on the left side with the remaining data on the right). The original call take a constant time (c) for the number of elements in the array (n) and each successive call takes one less than this as one element will be in a partition on its own.

```
cn+c(n-1)+c(n-2)+...+2c=c(n+(n-1)+(n-2)+...+2)
                        =c((n+1)(n/2)-1)
```

Expanding this gives us an n^2 function, hence giving us a complexity of $O(n^2)$.

Average Case:
Due to the inclusion of the helper method when implementing this sort, the average case follows that of the best case with the same proof holding. Again, this gives us a complexity of $O(nlog(n))$.

1.3 Merge Sort

Merge sort is again a more advanced sorting algorithm, that uses recursion to progressively sort and rebuild a data set until it's sorted. In this algorithm the data set is repeatedly split into smaller and smaller subsets of the data which can then be sorted. The sorted subsets are then recombined into progressively larger, sorted sets until the original set is reconstructed in a sorted order.

```
// Helper function to merge two halves of the array being sorted by the merge
//sort function
void merge(int arr[], int leftIndex, int middleIndex, int rightIndex)
{
    int leftArraySize = middleIndex - leftIndex + 1;
    int rightArraySize =  rightIndex - middleIndex;

    //Create two temporary arrays for each half
    int leftArray[leftArraySize];
    int rightArray[rightArraySize];

    //Copy the data to these temporary variables
    for (int i = 0; i < leftArraySize; i++)
```

```
    {
        leftArray[i] = arr[leftIndex + i];
    } //for
    for (int j = 0; j < rightArraySize; j++)
    {
        rightArray[j] = arr[middleIndex + 1+ j];
    } //for

    //Merge the temporary arrays back into original array
    int mergeIndex = leftIndex; // Initial index of merged subarray
    leftIndex = 0; // Initial index of first subarray
    rightIndex = 0; // Initial index of second subarray
    while (leftIndex < leftArraySize && rightIndex < rightArraySize)
    {
        if (leftArray[leftIndex] <= rightArray[rightIndex])
        {
            arr[mergeIndex] = leftArray[leftIndex];
            leftIndex++;
        } //if
        else
        {
            arr[mergeIndex] = rightArray[rightIndex];
            rightIndex++;
        } //else
        mergeIndex++;
    } //while

    //Copy any remaining elements from the left or right temporary arrays back
    //into array
    while (leftIndex < leftArraySize)
    {
        arr[mergeIndex] = leftArray[leftIndex];
        leftIndex++;
        mergeIndex++;
    } //while

    while (rightIndex < rightArraySize)
    {
        arr[mergeIndex] = rightArray[rightIndex];
        rightIndex++;
        mergeIndex++;
    } //while
} //merge

void merge_sort(struct darray* arr, int leftIndex, int rightIndex)
{
```

```
   if(leftIndex < rightIndex)
   {
      //Find the index of the middle of the array
      int middleIndex = (leftIndex + rightIndex) / 2;

      //Recursively sort the two halves of the list
      merge_sort(arr, leftIndex, middleIndex);
      merge_sort(arr, middleIndex + 1, rightIndex);

      //Merge the sorted lists using the helper function
      merge(arr, leftIndex, middleIndex, rightIndex);
   }
} //merge_sort
```

Assumed complexity: $O(nlog(n))$

Proof:
The proof of this complexity can be described relatively simply in terms of the algorithm itself, with the helper method running through the data set in its entirety a number of times (hence giving a $O(n)$ complexity) and the main algorithm itself splitting the data in half with each iteration (giving $O(log(n))$). This can be written more formally as...

$$T(n) = 2T(n/2) + n$$

Hence giving us a complexity of $O(nlog(n))$.
The divide and conquer approach with this algorithm, along with the always linear complexity of the helper function means that this algorithm retains this complexity in both the best and worst case with the proof above holding...

Best Case: $O(nlog(n))$

Worst Case: $O(nlog(n))$

Average Case: $O(nlog(n))$

1.4 Bubble Sort

The bubble sort algorithm works by repeatedly passing through an unsorted data set, comparing neighbouring items and swapping them if they are in the incorrect order. Despite being relatively easy to understand and implement the bubble sort is inefficient, particularly on larger data sets. The efficiency of the algorithm can be improved slightly by checking after each pass if a swap has occurred, preventing unnecessary passes through the data set once it is

already sorted. For this task I have implemented and analysed the first of these algorithms.

```
void bubble_sort(struct darray* arr)
{
  for (int i = 0; i < arr->size - 1; ++i)
  {
    for (int k = 0; k < arr->size - i - 1; ++k)
    {
      if (arr->cells[k] > arr->cells[k + 1])
      {
        int swapValue = arr->cells[k];
        arr->cells[k] = arr->cells[k + 1];
        arr->cells[k + 1] = swapValue;
      } //if
    } //for
  } //for
} //bubble_sort
```

Assumed complexity: $O(n^2)$

Proof:
The simplicity of this algorithm makes recognising the complexity relatively trivial but I will run through the proof for clarity...

As the algorithm above shows, the number of comparisons made on each pass reduces by 1 each time (with the inner loop being dependent on the value of the outer loop), hence giving us...

$$\begin{aligned}(n-1)+(n-2)+(n-3)+...+3+2+1\\ =n(n-1)/2\end{aligned}$$

This gives us an equation for the number of comparisons made in the order of n^2, hence giving us a complexity of $O(n^2)$.
As with the merge sort this algorithm holds this complexity in all cases with the same proof, with the only exception being when the optimization is made and the algorithm runs in the best case...

Best Case: $O(n^2)$ (or $O(n)$ using the optimised algorithm)

Worst Case: $O(n^2)$

Average Case: $O(n^2)$

2 Experimental Analysis

In this section we consider the question

> Under what conditions is it better to perform linear search rather than binary search?

To answer this question we must first make some assumptions about the nature of the question, notably the focus on the efficiency of the algorithm as oppose to the difficulty in implementing/understanding the algorithms themselves. Under these assumptions I have decided to focus my efforts on experimenting with the size and structure of the data set (in this case the dictionary being used).

2.1 Experimental Design

As described above, I have decided to vary the size and structure of the dictionary that will be passed to the search functions. I hope that changing these parameters will give the best results in comparing the conditions in which both algorithms perform best. To conduct this experiment I will create a script that will run the search algorithms with dictionaries of varying sizes and structures, displaying the time it takes to find the value for each run.

I have decided to test three sizes of dictionary:

- Small - 10 elements
- Medium - 1,000 elements
- Large - 10,000 elements

I have decided to test three methods of structuring the dictionary:

- Random
- Sorted (in order)
- Sorted (reverse order)

2.2 Experimental Results

	S	M	L	S	M	L	S	M	L
Linear Search	0.003s	0.008s	2.807s	0.002s	0.016s	3.211s	0.002s	0.0014s	2.111s
Binary (Insertion)	0.002s	0.011s	2.983s	0.002s	0.021s	4.524s	0.001s	0.0022s	4.329s
Binary (Quick)	0.002s	0.010s	3.012s	0.002s	0.008s	0.089s	0.003s	0.0014s	2.452s
Binary (Bucket)	—	—	—	—	—	—	—	—	—
Binary (Merge)	0.002s	0.006s	0.067s	0.001s	0.016s	0.045s	0.002s	0.0004s	0.051s
Binary (Bubble)	0.003s	0.012s	3.837s	0.002s	0.021s	4.769s	0.003s	0.0022s	4.331s

The results returned from the experiment that I conducted confirm the assumptions made about the relative complexities of the different algorithms.

Notably, that the algorithms with the worst complexities (the bubble and insertion sorts with a complexity of $O(n^2)$) performed dramatically worse than the other algorithms especially on larger data sets.

Similarly, the algorithms with the best complexities (the quick and merge sort with a complexity of $O(nlog(n))$) perform considerably better than the other algorithms.

Due to difficulties implementing the bucket sort algorithm I decided to leave the results out of the experiment, although we can use this experiment to estimate its usefulness.

3 Extending Experiment to Data Structures

We now extend our previously analysis to consider the question

> Under what conditions are different implementations of the dictionary data structure preferable?

4 Conclusions