EE4033 Algorithms

Programming Assignment #1

B11901110 陳璿吉

I. Implementation of Insertion Sort, Merge Sort, Quick Sort and Heap Sort

Most of the implementation is based on the pseudo-code taught in the course. The codes are given in the following figures.

```
// Insertsion sort method
void SortTool::InsertionSort(vector<int>& data)
{
    // Function : Insertion sort
    // TODO : Please complete insertion sort code here
    if(data.size() > 1)
    {
        int j, key;
        for( int i = 1 ; i < data.size() ; i++)
        {
            key = data.at(i);
            j = i - 1;
            while(j >= 0 && data.at(j) > key)
            {
                data.at(j + 1) = data.at(j);
                j = j - 1;
            }
            data.at(j + 1) = key;
        }
    }
}
```

Figure 1. Insertion Sort

```
// Sort subvector (Quick sort)
void SortTool::QuickSortSubVector(vector<int>& data, int low, int high, const int flag)
{

    // Function : Quick sort subvector
    // TODO : Please complete QuickSortSubVector code here
    // Hint : recursively call itself
    //        Partition function is needed
    // flag == 0 -> normal QS
    // flag == 1 -> randomized QS
    if(low < high)
    {
        int q;
        if(flag == 0)
            q = Partition(data, low, high);
        if(flag == 1)
            q = RandomizedPartition(data, low, high);

        QuickSortSubVector(data, low, q - 1, flag);
        QuickSortSubVector(data, q + 1, high, flag);
    }
}
```

Figure 2. `QuickSortSubVector` function, including randomized case

```cpp
int SortTool::RandomizedPartition(vector<int>& data, int low, int high)
{
    // Function : RQS's Partition the vector
    // TODO : Please complete the function
    srand(time(0));

    int i = low + (rand() % (high - low + 1)), temp;

    temp = data.at(high);
    data.at(high) = data.at(i);
    data.at(i) = temp;

    return Partition(data, low, high);
}
```

Figure 3. `RandomizedPartition` implemented using `rand()`

```cpp
int SortTool::Partition(vector<int>& data, int low, int high)
{
    // Function : Partition the vector
    // TODO : Please complete the function
    int i = low - 1, temp;
    for( int j = low ; j < high ; j++)
    {
        if(data.at(j) <= data.at(high))
        {
            i++;
            temp = data.at(j);
            data.at(j) = data.at(i);
            data.at(i) = temp;
        }
    }

    temp = data.at(high);
    data.at(high) = data.at(i + 1);
    data.at(i + 1) = temp;

    return i + 1;
}
```

Figure 4. `Partition` of Quick Sort

```cpp
// Merge sort method
void SortTool::MergeSort(vector<int>& data)
{
    MergeSortSubVector(data, 0, data.size() - 1);
}

// Sort subvector (Merge sort)
void SortTool::MergeSortSubVector(vector<int>& data, int low, int high)
{
    // Function : Merge sort subvector
    // TODO : Please complete MergeSortSubVector code here
    // Hint : recursively call itself
    //        Merge function is needed
    if(low < high)
    {
        int mid = (int)((low + high)/2);
        MergeSortSubVector(data, low, mid);
        MergeSortSubVector(data, mid + 1, high);
        Merge(data, low, mid, mid + 1, high);
    }
}
```

Figure 5. `MergeSortSubVector` function for Merge Sort

```cpp
// Merge
void SortTool::Merge(vector<int>& data, int low, int middle1, int middle2, int high)
{
    // Function : Merge two sorted subvector
    // TODO : Please complete the function
    int element_left, element_right, index_left, index_right;
    element_left = middle1 - low + 1;
    element_right = high - middle2 + 1;

    vector<int> left(element_left + 1, 0);
    vector<int> right(element_right + 1, 0);

    for( int i = 0 ; i < element_left ; i++)
        left.at(i) = data.at(low + i);

    for( int j = 0 ; j < element_right ; j++)
        right.at(j) = data.at(middle2 + j);

    left.at(element_left) = INT_MAX;
    right.at(element_right) = INT_MAX;

    index_left = 0;
    index_right = 0;
    for( int k = low ; k <= high ; k++)
    {
        if(left.at(index_left) <= right.at(index_right))
        {
            data.at(k) = left.at(index_left);
            index_left++;
        }
        else
        {
            data.at(k) = right.at(index_right);
            index_right++;
        }
    }
}
```

Figure 6. `Merge` function for Merge Sort

```cpp
// bottom-up style implementation of merge sort
void SortTool::BottomUpMergeSort(vector<int>& data)
{
  /*TODO :
    Implement merge sort in bottom-up style, in other words,
    without recursive function calls.
    Hint:
    1. Divide data to n groups of one data each group
    2. Iteratively merge each pair of 2 neighbor groups into one larger group
    3. Finally we obtain exactly one sorted group
  */
    int numGroup = data.size();
    int groupMem = 1;
    while(numGroup > 1)
    {
        int i = 0;
        while(i < data.size())
        {
            int start = i;
            int middle  = ((i + groupMem) < data.size()) ? (i + groupMem) : data.size();
            int end = ((i + groupMem * 2) < data.size()) ? (i + groupMem * 2) : data.size();

            vector<int> left(data.begin() + start, data.begin() + middle);
            vector<int> right(data.begin() + middle, data.begin() + end);

            vector<int> merged(left.size() + right.size());

            int left_index = 0, right_index = 0;

            while(left_index < left.size() && right_index < right.size())
            {
```

```cpp
                if(left[left_index] <= right[right_index])
                {
                    merged[left_index + right_index] = left[left_index];
                    left_index++;
                }
                else
                {
                    merged[left_index + right_index] = right[right_index];
                    right_index++;
                }
            }

            if(left_index < left.size())
            {
                for( int j = left_index ; j < left.size() ; j++)
                    merged[j + right_index] = left[j];
            }

            if(right_index < right.size())
            {
                for( int j = right_index ; j < right.size() ; j++)
                    merged[left_index + j] = right[j];
            }

            for( int j = 0 ; j < groupMem * 2 ; j++)
                data[i + j] = merged[j];

            i = i + groupMem * 2;

        }

        numGroup = (numGroup % 2 == 0) ? (numGroup / 2) : ((numGroup/2) + 1);
        groupMem = groupMem * 2;
    }
```

Figure 7. BottomUpMergeSort

```cpp
//Max heapify
void SortTool::MaxHeapify(vector<int>& data, int root)
{
    // Function : Make tree with given root be a max-heap if both right and left sub-tree are max-heap
    // TODO : Please complete max-heapify code here
    int left, right, heap_max_index, temp;
    left = 2 * root + 1;
    right = 2 * root + 2;

    if(left < heapSize && data.at(left) > data.at(root))
        heap_max_index = left;
    else
        heap_max_index = root;

    if(right < heapSize && data.at(right) > data.at(heap_max_index))
        heap_max_index = right;

    if(heap_max_index != root)
    {
        temp = data.at(heap_max_index);
        data.at(heap_max_index) = data.at(root);
        data.at(root) = temp;

        MaxHeapify(data, heap_max_index);
    }
}
```

Figure 8. `MaxHeapify` function for Heap Sort

```cpp
//Build max heap
void SortTool::BuildMaxHeap(vector<int>& data)
{
    heapSize = data.size(); // initialize heap size
    // Function : Make input data become a max-heap
    // TODO : Please complete BuildMaxHeap code here
    for(int i = ((int)(data.size() / 2) - 1) ; i >= 0 ; i--)
        MaxHeapify(data, i);
}
```

Figure 9. `BuildMaxHeap` function for Heap Sort

II. Analysis of Insertion Sort, Merge Sort, Quick Sort and Heap Sort

The running time and memory usage of Insertion Sort, Merge Sort, Bottom-Up Merge Sort, Quick Sort, Randomized Quick Sort and Heap Sort is given by the following tables. The code is tested using EDA Union Server on MobaXterm v23.6.

Input Size	Insertion Sort		Merge Sort		Quick Sort		Heap Sort	
	CPU Time (ms)	Memory (kB)	CPU Time (ms)	Memory (kB)	CPU Time (ms)	Memory (kB)	CPU Time (ms)	Memory (kB)
1000.case1	0.858	5908	0.643	5908	0.304	5908	0.476	5908
1000.case2	0.133	5908	0.446	5908	2.149	5908	0.396	5908
1000.case3	2.31	5908	0.44	5908	2.23	5908	0.4	5908
2000.case1	5.276	5908	1.007	5908	0.553	5908	0.87	5908
2000.case2	0.138	5908	0.778	5908	6.174	5912	0.706	5908
2000.case3	6.858	5908	0.776	5908	7.348	5908	0.704	5908
4000.case1	9.423	5908	1.977	5908	0.93	5908	1.799	5908
4000.case2	0.106	5908	1.456	5908	14.968	6040	1.264	5908
4000.case3	12.249	5908	1.429	5908	15.858	5908	1.412	5908
8000.case1	17.582	6060	2.477	6060	1.274	6060	2.911	6060
8000.case2	0.156	6060	1.296	6060	38.831	6436	2.158	6060
8000.case3	28.262	6060	2.413	6060	38.586	6188	2.545	6060
16000.case1	48.725	6060	3.583	6060	2.41	6060	3.588	6060
16000.case2	0.145	6060	2.177	6060	138.548	6936	2.367	6060
16000.case3	95.43	6060	2.837	6060	139.007	6432	3.541	6060
32000.case1	185.964	6192	6.253	6192	3.441	6192	4.241	6192
32000.case2	0.148	6192	4.332	6192	546.585	8004	3.604	6192
32000.case3	370.897	6192	4.376	6192	530.44	6988	4.062	6192
1000000.case1	187877	12148	192.427	14008	89.922	12148	194.344	12148
1000000.case2	1.338	12148	108.879	14008	529111	72468	118.731	12148
1000000.case3	376770	12148	112.507	14008	361901	33016	116.608	12148

Table 1. Comparison of Insertion Sort, Merge Sort, Quick Sort and Heap Sort

Input Size	Top-Down Merge Sort		Bottom-Up Merge Sort	
	CPU Time (ms)	Memory (kB)	CPU Time (ms)	Memory (kB)
1000.case1	0.643	5908	0.63	5908
1000.case2	0.446	5908	0.45	5908
1000.case3	0.44	5908	0.446	5908
2000.case1	1.007	5908	1.132	5908
2000.case2	0.778	5908	0.748	5908
2000.case3	0.776	5908	0.745	5908
4000.case1	1.977	5908	1.138	6052
4000.case2	1.456	5908	0.82	6052
4000.case3	1.429	5908	0.828	6052
8000.case1	2.477	6060	3.756	6060
8000.case2	1.296	6060	2.147	6060
8000.case3	2.413	6060	1.478	6060
16000.case1	3.583	6060	2.498	6216
16000.case2	2.177	6060	1.442	6216
16000.case3	2.837	6060	2.342	6216
32000.case1	6.253	6192	5.648	6380
32000.case2	4.332	6192	3.447	6380
32000.case3	4.376	6192	3.203	6380
1000000.case1	192.427	14008	240.654	21828
1000000.case2	108.879	14008	177.275	21828
1000000.case3	112.507	14008	176.844	21828

Table 2. Comparison of Top-Down Merge Sort and Bottom-Up Merge Sort

From Table 2, we observe that when the input size gets really large, Bottom-Up Merge Sort takes significantly more time and space than Top-Down Merge Sort. It is mainly because there are more iterations and allocation of temporary array in the Top-Down approach.

Input Size	Quick Sort		Randomized Quick Sort	
	CPU Time (ms)	Memory (kB)	CPU Time (ms)	Memory (kB)
1000.case1	0.304	5908	3.278	5908
1000.case2	2.149	5908	3.057	5908
1000.case3	2.23	5908	4.058	5908
2000.case1	0.553	5908	5.326	5908
2000.case2	6.174	5912	7.053	5908
2000.case3	7.348	5908	7.389	5908
4000.case1	0.93	5904	9.086	5908
4000.case2	14.968	6028	10.318	5908
4000.case3	15.858	5908	9.691	5908
8000.case1	1.274	6060	12.758	6060
8000.case2	38.831	6432	12.035	6060
8000.case3	38.586	6188	12.16	6060
16000.case1	2.41	6056	18.855	6060
16000.case2	138.548	6932	16.145	6060
16000.case3	139.007	6428	17.381	6060
32000.case1	3.441	6188	34.199	6192
32000.case2	546.585	7996	33.231	6192
32000.case3	530.44	6988	30.895	6192
1000000.case1	89.922	12144	1032.68	12148
1000000.case2	529111	72472	996.115	12148
1000000.case3	361901	33012	955.969	12148

Table 3. Comparison of Quick Sort and Randomized Quick Sort

Judging from the table, we can see that Randomized Quick Sort performs worse in average cases, but significantly outperforms Quick Sort in the best case and the worst case. In addition, Randomized Quick Sort uses less space than Quick Sort.

The following graphs illustrate the performance of Insertion Sort, Merge Sort, Quick Sort, Randomized Quick Sort and Heap sort in different cases.

1. Average Case

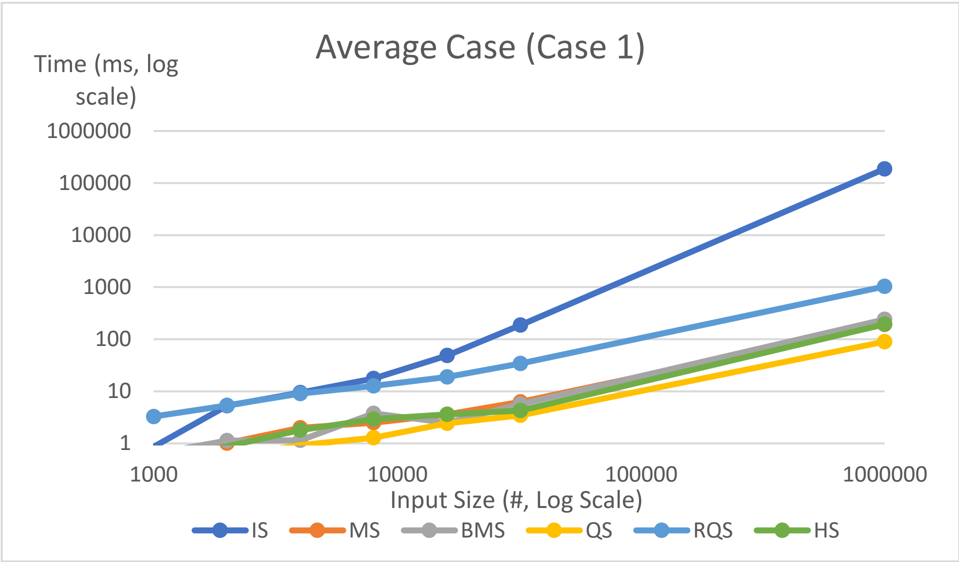

Figure 9. Performance of different sorting algorithms on average case

As we learn from the course, on average, the running time of insertion sort is $T(n) = \Theta(n^2)$, since two for loops are executed. On the other hand, the remaining 5 sorting algorithms have average running time $T(n) = \Theta(n \lg n)$. Therefore, insertion sort takes more time to sort an array, and such difference is more significant when the number of elements becomes large. The rest of the sorting algorithms take approximately the same time to sort the given arrays.

We can also observe the performance of these 5 algorithms based on the slope of the lines. The slope m is calculated using the following formula:

$$m = \frac{\log T_2 - \log T_1}{\log n_2 - \log n_1}, \tag{1}$$

where (n_1, T_1) and (n_2, T_2) represents the number of elements and running time at two points. What does the slope mean? It actually represents the order of an algorithm. For example, suppose $T(n) = n^2$, then $\log T(n) = 2\log n$ and $m = 2$. We use $n_1 = 32000$ and $n_2 = 1000000$ to calculate the slope, which is given in the following table:

Sorter	m
Insertion Sort	2.01
Merge Sort	1.00
Bottom-Up Merge Sort	1.09
Quick Sort	0.95
Randomized Quick Sort	0.99
Heap Sort	1.11

Table 4. Order of Different Sorting Algorithms: Average Case

From Table 4, we observe that Insertion sort has $m = 2.01$. This is roughly consistent with the theoretical running time $T(n) = \Theta(n^2)$. On the other hand, the rest sorting algorithms have running time $T(n) = \Theta(n \lg n)$. Hence m should be approximately 1, and Table 4 confirms such deduction.

2. Best Case (Sorted Array as Input)

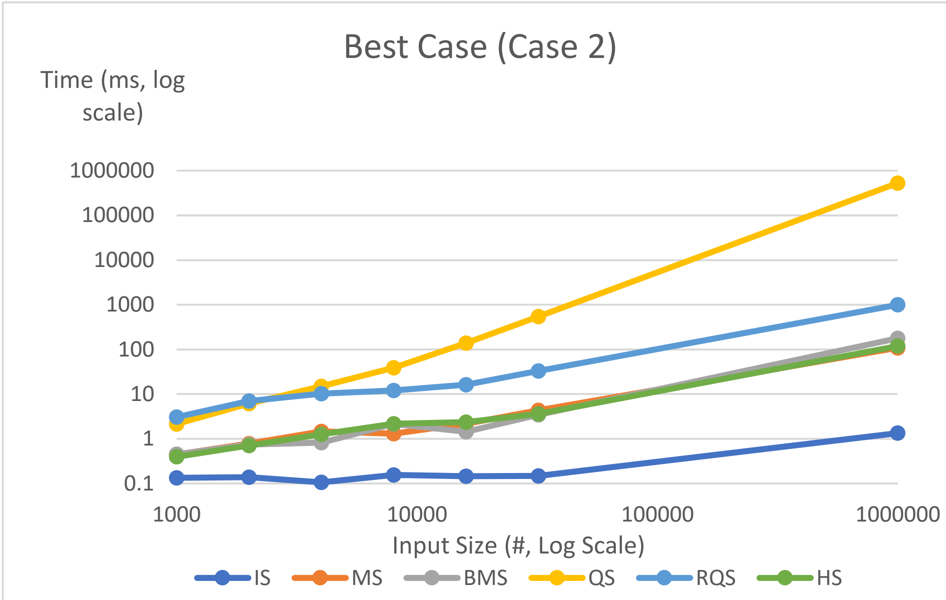

Figure 10. Performance of different sorting algorithms on best case

The "Best Case" here refers to a sorted array as input. From Figure 10 we observe that insertion sort takes the least time among the 5 algorithms. When the input array is sorted, the insertion sort has a running time $T(n) = \Omega(n)$, since the inner while loop doesn't execute and swap numbers. In this case, the insertion sort is equivalent as a linear scan. On the other hand, quick sort takes the most time to sort the input array. While designing the quick sort algorithm, our implementation chooses the last index as the pivot point. This dramatically influenced the performance of quick sort since the recursion tree is unbalanced, with one side having $(n-1)$ elements and the other having 1, which is the pivot point itself. The running time of the original quick sort then became $T(n) = O(n^2)$. A sorted array is the worst case for the original quick sort.

To improve the performance of quick sort, we implemented a randomized quick sort algorithm which chooses the pivot point randomly. With such modification, the running time of randomized quick sort became $T(n) = \Omega(n \lg n)$. As Figure 10 shows, the randomized quick sort runs asymptotically as fast as merge sort and heap sort.

We calculate the order of each algorithm using (1), which gives the following table:

Sorter	m
Insertion Sort	0.64
Merge Sort	0.94
Bottom-Up Merge Sort	1.14
Quick Sort	2.00
Randomized Quick Sort	0.99
Heap Sort	1.11

Table 5. Order of Different Sorting Algorithms: Best Case

From Table 5, it is noteworthy that Insertion Sort has $m = 0.64$, which is significantly less than the theoretical value $m = 1$. The variation of running time in each execution may be the reason. In addition, the behavior of Insertion Sort in the best case is simply scanning through the array, such operation may be faster than swapping elements. As for Quick Sort, we can see that $m = 2$, which confirms that Quick Sort runs in $O(n^2)$ time complexity in this case. The remaining sorting algorithms performs approximately the same with m being close to 1, but Bottom-up Merge Sort and Heap Sort are slightly slower than Merge Sort and Randomize Quick Sort.

3. Worst Case (Reversed Array as Input)

Figure 11. Performance of different sorting algorithms on worst case

The "worst case" here refers to a reversed array as an input. From Figure 11, we conclude that the insertion sort and the quick sort takes the most time to finish the task. For the insertion sort, since the array is reversed, the inner while loop must swap the numbers from the rightmost position all the way to the left, so the running time of the insertion sort is $T(n) = O(n^2)$. The reason why the "quick" sort takes as much time as the insertion sort is given in the analysis of "best case" part. A reversed array is also the worst case for the quick sort since the recursion tree is unbalanced, with one side having $(n-1)$ elements and the other side having 1 element, which is the smallest element. In this case, the quick sort has running time $T(n) = O(n^2)$

Similar to the analysis of "best case," the performance of quick sort can be improved by choosing the pivot point randomly. The running time of randomized quick sort is $T(n) = O(n \lg n)$. Overall, the merge sort and the heap sort runs are typically the fastest sorting algorithms. Their performances don't vary significantly with the input, and have consistent running time $T(n) = \Theta(n \lg n)$.

We calculate the order of each algorithm using (1), which gives the following table:

Sorter	m
Insertion Sort	2.01
Merge Sort	0.94
Bottom-Up Merge Sort	1.17
Quick Sort	1.90
Randomized Quick Sort	1.00
Heap Sort	0.98

Table 6. Order of Different Sorting Algorithms: Worst Case

In the worst case, Insertion Sort has $m = 2.01$, which is consistent with the theoretical running time $\Theta(n^2)$. For Quick Sort, it is similar to the previous case, with m being close to 2. Bottom-Up Merge Sort performs worse than Merge Sort owing to more iterations. The remaining sorting algorithms have $m \approx 1$ and perform asymptotically the same.