CS 201 - Homework 2 Report

Mehmet Akif Şahin - 22203673

15 November 2023

In this report several sorting algorithms will be analyzed in order to satisfy the needs of Bilkent University Library.

Computer Specifications

Intel i7-11800H
16GB Ram
Ubuntu 22.04 64-bit
Compiler: g++ 11.4.0

Task 1: Which Algorithm?

Bilkent Library is suggested to use Merge Sort and Quick Sort but one of the staff insists on using Bubble Sort. Let's start by analyzing Bubble Sort.

Bubble Sort

Algorithm for Bubble Sort

1. Traverse from left to right while comparing adjacent elements and swapping if they are in the wrong order. This way the biggest element goes to the right of the list.

2. Continue doing step 1 until no swaps occur in a one sort pass.

C++ Implementation of Bubble Sort

```
1  void bubbleSort(int *& arr, int size) {
2    bool swapped = true;
3    int temp;
4    for (int i = 0; i < size && swapped; i++){
5      swapped = false;
6      for (int j = 1; j < size - i; j++) {
7        if (arr[j-1] > arr[j]) {
8          swapped = true;
9          temp = arr[j-1];
10         arr[j-1] = arr[j];
11         arr[j] = temp;
12       }
13     }
14   }
15 }
```

Analysis for Bubble Sort Bubble Sort uses constant $O(1)$ extra space to sort the array and in the worst case makes

$$\sum_{i=0}^{size-1} \sum_{j=1}^{size-i-1} 1 = \sum_{i=0}^{size-1} (size - i - 1) = \frac{size(size-1)}{2} = O(size^2) = O(n^2)$$

comparisons. In the best case (when the input array is already sorted) Bubble Sort makes just a one sort pass and determines that the array is sorted. Bubble Sort has $O(n)$ time complexity in the best case.

Merge Sort

Algorithm for Merge Sort

1. Divide the list into 2 parts.
2. Merge Sort each part individually.
3. Merge 2 parts.

Merge : Merging is the operation to unite 2 lists while keeping the order of the elements.

Implementing a merge function is the crucial part of implementing merge sort.

Algorithm for Merge Operation

1. Create a new list with size equal to the sum of the sizes of the lists that are going to be merged.
2. Go through both the lists, comparing the elements. Take the smaller one and put it in the new list, until reaching end of one of the lists.
3. If there are still elements in one of the lists, add them to the new list.
4. The new list is the merge of the two lists.

C++ Implementation for Merge Operation

```
1  void merge(int*& arr, int left, int mid, int right) {
2    int* temp = new int[right - left + 1];
3    int index1 = left;
4    int index2 = mid+1;
5    int index = 0;
6    while (index1 <= mid && index2 <= right) {
7      if (arr[index1] <= arr[index2]) temp[index++] = arr[index1++];
8      else temp[index++] = arr[index2++];
9    }
10   while (index1 <= mid) temp[index++] = arr[index1++];
11   while (index2 <= right) temp[index++] = arr[index2++];
12   for (index = 0; index < right - left + 1; index++)
13     arr[left + index] = temp[index];
14   delete[] temp;
15 }
```

Analysis for Merge Operation Merge operation uses a array of size $(right - left)$ meaning $O(right - left) = O(n)$ extra space. And in the worst case makes

$$min(right - mid, mid - left + 1) = O(right - left) = O(n)$$

comparisons.

C++ Implementation for Merge Sort

```
1 void mergeSort (int *& arr, int left, int right) {
2   if (left < right) {
3     int mid = (left + right) / 2;
4     mergeSort(arr, left, mid);
5     mergeSort(arr, mid + 1, right);
6     merge(arr, left, mid, right);
7   }
8 }
```

Analysis for Merge Sort Let $T(n)$ denote the number of comparisons made by merge sort with the value $n = right - left$ denoting the size. Then $T(1) = O(1)$ and

$$T(n) = 1 + 2T(\frac{n}{2}) + O(n) = O(n) + 2T(\frac{n}{2})$$

the $O(n)$ at the end denotes the complexity of merge operation with $n = right - left$.

$$T(n) = O(n) + 2(O(\frac{n}{2}) + 2T(\frac{n}{4})) = \sum_{i=1}^{k} 2^{k-1}O(\frac{n}{2^{k-1}}) + 2T(\frac{n}{2^k}) = kO(n) + 2T(\frac{n}{2^k})$$

$$T(n) = \log_2 nO(n) + 2T(1) = \log_2 nO(n) + 2O(1) = O(n \log_2 n), \text{ for } k = \log_2 n$$

Time complexity of merge sort is $O(n \log n)$. Memory complexity of merge sort can be found in a similar fashion. Let $M(n)$ denote the memory complexity of merge sort. Then M(1) = 1 and,

$$M(n) = 2M(\frac{n}{2}) + O(n)$$

using the same method as before it can be concluded that

$$M(n) = O(n \log n)$$

Merge Sort uses extra $O(n \log n)$ total space.

Quick Sort

Algorithm for Quick Sort

1. Select a pivot.
2. Partition the list according to the pivot.
3. Quick sort the 2 parts that are divided by the pivot.

Partition : Partitioning is the process of rearranging elements in an list so that elements less than or equal to a chosen pivot are on one side, and elements greater than the pivot are on the other side.

Partition is the backbone of the Quick Sort algorithm.

Partition Algorithm

1. Choose the pivot to be the first item.
2. Set up 2 index variables; **firstUnknown** to refer to the first element of the not yet partitioned part of the list, **lastS1** to refer to the last element of the first part of the partitioned list.
3. Iterate over the list while incrementing the firstUnknown variable; if the current element is lesser than the pivot, swap the element with the **lastS1** + **1**'th element and increment **lastS1** by one.
4. Swap the pivot (was chosen as the first element) with the last element of the first part of the partition, **lastS1**'th element.
5. Now the partition is complete. The pivot is at the index **lastS1**.

C++ Implementation for Partition

```
1  void partition(int*& arr, int left, int right, int& pivotIndex) {
2    int pivot = arr[left];
3    int unknown = left + 1;
4    int s1end = left;
5
6    for (; unknown <= right; unknown++){
7      if (arr[unknown] < pivot) {
8        swap(arr[unknown], arr[++s1end]);
9      }
10   }
11   pivotIndex = s1end;
12   swap(arr[left], arr[pivotIndex]);
13 }
```

Analysis for Partition Partition uses constant $O(1)$ extra space to partition the array and it makes

$$\text{(loop condition)} + \text{(if statement)} = (right - left + 1) + (right - left)$$

$$2 * right - 2 * left + 1 = O(n) \text{ for } n = right - left$$

comparisons.

C++ Implementation for Quick Sort

```
1 void quickSort (int *& arr, int left, int right) {
2   if (left < right) {
3     int pivotIndex;
4     partition(arr, left, right, pivotIndex);
5     quickSort(arr, left, pivotIndex - 1);
6     quickSort(arr, pivotIndex + 1, right);
7   }
8 }
```

Analysis for Quick Sort Quick Sort uses $O(1)$ extra memory to sort the elements of the array. For the time complexity, let $T(n)$ denote the time complexity for $n = right - left$ and $T(n) = O(1)$. In the worst case (the case which there are no elements in one of the parts of the list after partition every time)

$$T(n) = (\text{complexity of the partition}) + T(n-1) = O(n) + T(n-1)$$

$$T(n) = O(n) + O(n-1) + O(n-2) + T(n-3) = \sum_{i=1}^{k} O(n-i+1) + T(n-k)$$

$$T(n) = O(n^2) + T(1) = O(n^2) \text{ for } k = n-1$$

Quick Sort has $O(n^2)$ time complexity. In the average case, wishfully assuming that in every partition the pivot takes place near the middle of the list,

$$T(n) = (\text{complexity of the partition}) + 2T(\frac{n}{2}) = O(n) + 2T(\frac{n}{2})$$

$$T(n) = O(n \log n) \qquad \text{(similar to the case with the merge sort)}$$

has $O(n \log n)$ time complexity. The advantage of Quick Sort over Merge Sort is the memory complexity.

Empirical Results for The Complexities of Sorting Algorithms

	Time			Memory		
Array Size	Bubble Sort	Merge Sort	Quick Sort	Bubble Sort	Merge Sort	Quick Sort
8	2e-07	9e-07	3e-07	0	24	0
16	6e-07	1e-06	5e-07	0	64	0
32	2.5e-06	1.5e-06	1.3e-06	0	160	0
64	8.3e-06	3.3e-06	2.6e-06	0	384	0
128	2.91e-05	7.6e-06	5.8e-06	0	864	0
256	0.0001028	1.58e-05	1.29e-05	0	2048	0
512	0.0003847	3.64e-05	2.85e-05	0	4608	0
1024	0.0014685	8.35e-05	6.63e-05	0	10240	0
2048	0.0057176	0.000193	0.0001434	0	22528	0
4096	0.0227771	0.0004237	0.0003104	0	49152	0

Table 1: Time Elapsed & Total Dynamically Allocated Extra Memory to Sort a Random Array (Average of 10 Experiments) (1 Memory = 1 integer in heap = 4 bytes in the system used in this experiment)

	Time			Memory		
Array Size	Bubble Sort	Merge Sort	Quick Sort	Bubble Sort	Merge Sort	Quick Sort
8	1e-07	6e-07	3e-07	0	24	0
16	4e-07	7e-07	6e-07	0	64	0
32	3e-07	1.3e-06	1.3e-06	0	160	0
64	1e-07	2.6e-06	4.2e-06	0	384	0
128	5e-07	5.2e-06	1.38e-05	0	896	0
256	4e-07	1.13e-05	5.11e-05	0	2048	0
512	1e-06	2.49e-05	0.0001933	0	4608	0
1024	2.1e-06	5.19e-05	0.0007414	0	10240	0
2048	4e-06	0.0001128	0.002955	0	22528	0
4096	8.4e-06	0.0002448	0.0115709	0	49152	0

Table 2: Time Elapsed & Total Dynamically Allocated Extra Memory to Sort a Sorted Array (Average of 10 Experiments) (1 Memory = 1 integer in heap = 4 bytes in the system used in this experiment)

	Time			Memory		
Array Size	Bubble Sort	Merge Sort	Quick Sort	Bubble Sort	Merge Sort	Quick Sort
8	5e-07	5e-07	1e-07	0	24	0
16	3e-07	5e-07	6e-07	0	64	0
32	1.9e-06	1.3e-06	1.9e-06	0	160	0
64	7e-06	2.4e-06	6.1e-06	0	384	0
128	2.72e-05	5.4e-06	2.19e-05	0	896	0
256	0.0001085	1.13e-05	8.12e-05	0	2048	0
512	0.0004268	2.55e-05	0.0003126	0	4608	0
1024	0.0017228	5.26e-05	0.0012501	0	10240	0
2048	0.0068055	0.0001122	0.0049484	0	22528	0
4096	0.0274753	0.0002456	0.0196862	0	49152	0

Table 3: Time Elapsed & Total Dynamically Allocated Extra Memory to Sort a Reverse Sorted Array (Average of 10 Experiments) (1 Memory = 1 integer in heap = 4 bytes in the system used in this experiment)

from - to	Bubble Sort	Merge Sort	Quick Sort
8 - 16	3	1.11111	1.66667
16 - 32	4.16667	1.5	2.6
32 - 64	3.32	2.2	2
64 - 128	3.50602	2.30303	2.23077
128 - 256	3.53265	2.07895	2.22414
256 - 512	3.74222	2.3038	2.2093
512 - 1024	3.81726	2.29396	2.32632
1024 - 2048	3.8935	2.31138	2.1629
2048 - 4096	3.98368	2.19534	2.16457

Table 4: Ratios of adjacent columns for Table 1 (Random)

Analysis for the Empirical Results of The Sorting Algorithms

Time Complexity

Firstly, the cases with fewer sizes are not a good indicator for complexities because when n is small, the coefficients which are neglected in the big-oh notation has significant effect on the time it takes to sort.

Starting with randomly created arrays, it is clear in the bottom rows that the theoretical complexity of bubble sort and quick & merge sort is different (Table 1). And if a table of the ratios of adjacent rows is made, which indicates the increase of the time spent with respect to input doubling, it is seen that bubble sorts time to sort quadruples as the input doubles and it is a little more than 2 for quick & merge sort (Table 4). This implies that complexity of bubble sort is $O(n^2)$ and the complexity for quick & merge sort is near $O(n)$. These observations do not conflict with the theoretical complexity.

$O(n^2)$ behavior of quick sort in the worst case can be observed in Table 2-5 & 3-6, it quadruples every time input doubles. $O(n)$ behavior of the bubble sort in the best case can be observed in Table 2. The stability of Merge Sort is exemplified across all six tables, regardless of the input array's initial condition. This constant stability is a strength of the Merge Sort algorithm.

All conclusions above can also be observed from the plots (Figures 1-2-3).

Regarding to the time complexity, my advice for the Bilkent University library is to use Merge Sort because of its stability when there are no constraints on the input.

from - to	Bubble Sort	Merge Sort	Quick Sort
8 - 16	4	1.16667	2
16 - 32	0.75	1.85714	2.16667
32 - 64	0.333333	2	3.23077
64 - 128	5	2	3.28571
128 - 256	0.8	2.17308	3.7029
256 - 512	2.5	2.20354	3.78278
512 - 1024	2.1	2.08434	3.83549
1024 - 2048	1.90476	2.17341	3.9857
2048 - 4096	2.1	2.17021	3.9157

Table 5: Ratios of adjacent columns for Table 2 (Sorted)

from - to	Bubble Sort	Merge Sort	Quick Sort
8 - 16	0.6	1	6
16 - 32	6.33333	2.6	3.16667
32 - 64	3.68421	1.84615	3.21053
64 - 128	3.88571	2.25	3.59016
128 - 256	3.98897	2.09259	3.70776
256 - 512	3.93364	2.25664	3.84975
512 - 1024	4.03655	2.06275	3.99904
1024 - 2048	3.95026	2.13308	3.9584
2048 - 4096	4.03722	2.18895	3.9783

Table 6: Ratios of adjacent columns for Table 3 (Reverse Sorted)

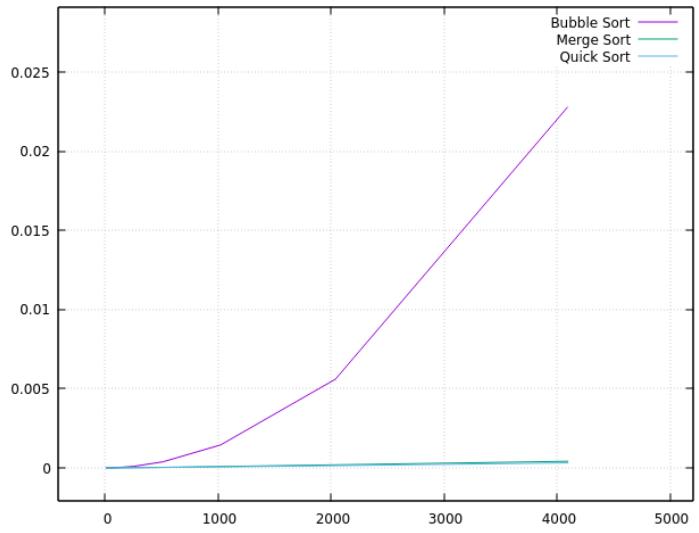

Figure 1: Plot for Table 1 (X-axis for Array Size, Y-axis for Time in Seconds)

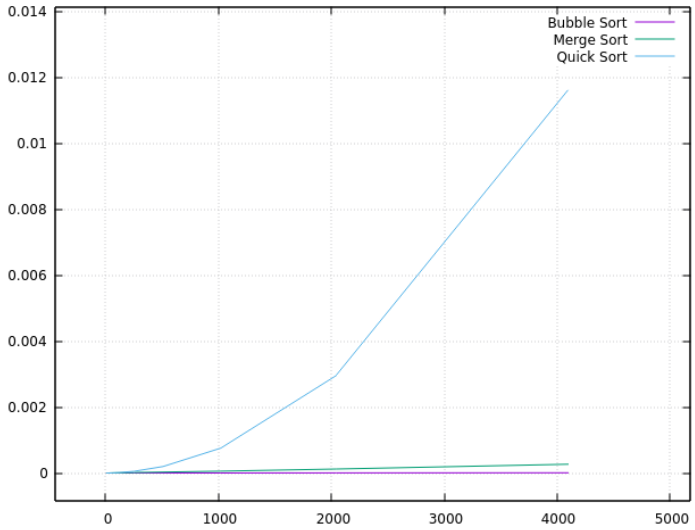

Figure 2: Plot for Table 2 (X-axis for Array Size, Y-axis for Time in Seconds)

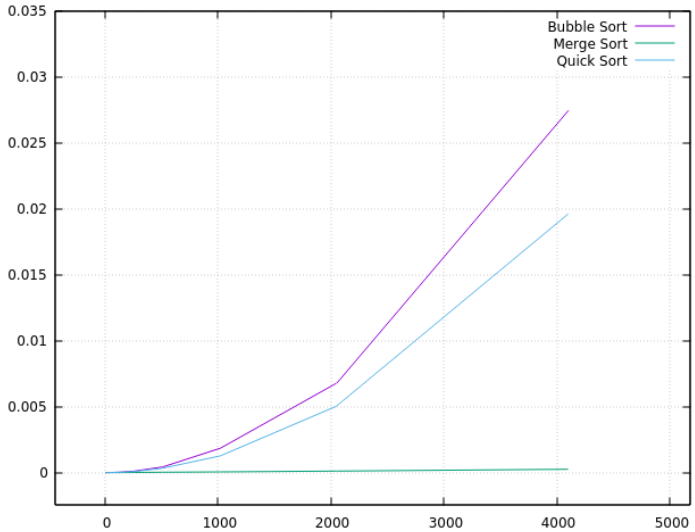

Figure 3: Plot for Table 3 (X-axis for Array Size, Y-axis for Time in Seconds)

Space Complexity

It can be seen from the graphs that the only algorithm that dynamically allocates extra memory for sorting is merge sort (Table 1-2-3). Furthermore it is seen that allocated total extra memory is equal to $n * \log n$ for $n = ArraySize$. If there are memory limitations, Bilkent Library should consider using bubble/merge sort.

Task 2: New Insights

Library authority informs that data is almost sorted. In this situation there is no need for algorithms for sorting the data all from scratch. Merge sort will do a good job on the time complexity but will use extra memory. Quick sort will do a bad job because almost sorted arrays are similar to the worst case and time complexity will be $O(n^2)$. Bubble sort will do a good job similar to merge sort but will use nearly zero extra memory. So my suggestion to Bilkent Library authority will be to use Bubble Sort. In Table 7, Quick Sort doesn't perform well, but Bubble Sort and Merge Sort are doing a great job.

Array Size	Bubble Sort	Merge Sort	Quick Sort
8	4e-07	1.5e-06	2e-07
16	3e-07	7e-07	6e-07
32	8e-07	1.4e-06	1.5e-06
64	2e-06	2.9e-06	5.5e-06
128	7.4e-06	6.1e-06	2.21e-05
256	2.42e-05	1.24e-05	9.04e-05
512	8.66e-05	2.77e-05	0.0003735
1024	0.0003327	5.83e-05	0.0015121
2048	0.0010533	0.000124	0.0059415
4096	0.00423	0.0002671	0.0232116

Table 7: Time Elapsed to Sort a Reverse Sorted Array (Average of 10 Experiments)

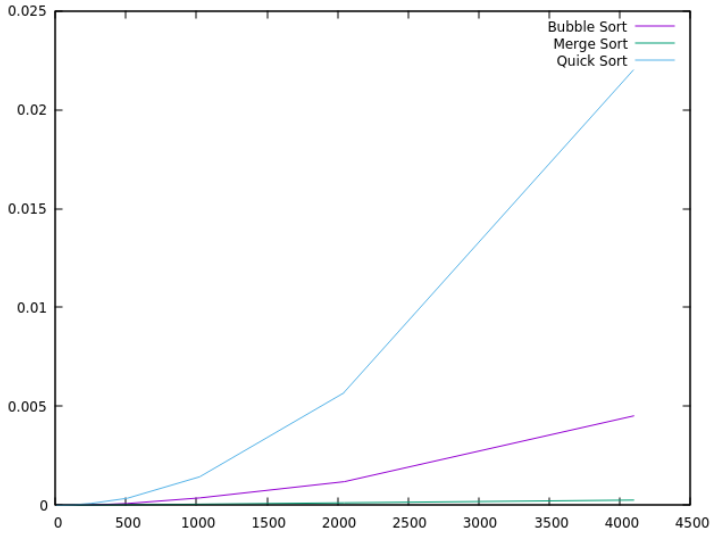

Figure 4: Plot for Table 7 (X-axis for Array Size, Y-axis for Time in Seconds)