Discipline Core

ALGORITHMS

CS4003

Assignment 1

Sheikh Muhammed Tadeeb (AU19B1014)

❖ Problem Statement:

Sorting is ordering a list of objects. Using comparisons to sort items is useful for nearly every algorithmic problem. In this exercise, you will implement and compare sorting algorithms and measure their efficiency in terms of swaps and comparisons. Finally, you will be commenting on the complexity trade-off for all the algorithms.

❖ Theory:

Sorting algorithms are ways to organize items of data structure based on certain rules. These algorithms can be used to organize messy data and make it easier to use. Each algorithm is having its own speed, uses, advantages, and disadvantages which are checked below.

A few examples of sorting algorithms:

- **Bubble sort:** This algorithm repeatedly proceeds through the list, comparing pairs of adjacent items and exchanging their positions if they are in the wrong order. The algorithm passes through the list in that way until the entire list has been sorted.
- **Insertion sort:** This algorithm starts by putting the first two items in order and then compares the third item with the second one, swapping positions if necessary and repeats that action with the first item. Subsequent items subjected to the same process often don't have to be moved far through the sorted items.
- **Selection sort:** Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.
- **Quick sort:** This algorithm selects a random item in the list, compares all the other items to it and organizes them into those that belong before the selected item and those that belong after it. That means that none of the items have to be compared with those in the other group again. The method proceeds by selecting random items within those two groups of items and repeating the process. Eventually, some other method such as the insertion algorithm does the final sorting.
- **Merge sort:** The Merge Sort function repeatedly divides the array into two halves until we reach a stage where we try to perform Merge Sort on a subarray of size 1

❖ C++ Code:

```
1.
2.  #include <iostream>
3.  using namespace std;
4.
5.  class emp{
6.        public:
7.              int empid;
8.              int salary;
9.              emp* next;
10. };
11.
12. class BubSort
13. {
14.       public:
15.             BubSort(){};
16.       protected:
17.             // Declaration of member functions
18.           emp* linkList_init();
19.           void display(emp* head);
20.           void SortedDisplay(emp* val);
21.           void bubble_sort(emp* head);
22.           void selection_sort(emp* head);
23.           void insertion_sort(emp* head);
24.           void quicksort(emp *start,emp *end);
25.           emp* partition(emp *start,emp* end);
26.           void merge_sort(int arr[]);
27.           void merge(int L, int R, int arr[]);
28.
29.          // Calling all the member functions in call() function.
30.           void call(){
31. //            emp* head;
32. //            head = linkList_init();
33. //            display(head);
34.               // ********************** Note here we will uncomment one at
    a time and run **********************
35. //            bubble_sort(head);
36. //            selection_sort(head);
37. //                  insertion_sort(head);
38. //                  emp* start = head;
39. //                  emp* end;
40. //                  while(start != NULL){
41. //                        end = start;
42. //                        start = start->next;
43. //                  }
44. //                  quicksort(head,end);
```

```
45. //                  SortedDisplay(head);
46.             }
47. };
48.
49. // Function to divide or break the list
50. // and conquer the problem.
51. emp* BubSort :: partition(emp *start,emp* end){
52.       // Base Case
53.       if(start == end || start == NULL || end == NULL){
54.             return start;
55.       }
56.       int pivot = end->salary;
57.       int pivot2 = end->empid;
58.       emp* low = start;
59.       emp* prev = start;
60.       while(start != end ){
61.                   if(start->salary < pivot){
62.                         prev = low;
63.                         int temp = low->salary;
64.                         low->salary = start->salary;
65.                         start->salary = temp;
66.                         temp = low->empid;
67.                         low->empid = start->empid;
68.                         start->empid = temp;
69.                         low = low->next;
70.                   }else if(start->salary == pivot && start->empid <
   pivot2){
71.                         int temp = low->salary;
72.                         low->salary = start->salary;
73.                         start->salary = temp;
74.                         temp = low->empid;
75.                         low->empid = start->empid;
76.                         start->empid = temp;
77.                         low = low->next;
78.             }
79.             start = start->next;
80.       }
81.       int temp1 = low->salary;
82.       low->salary = pivot;
83.       end->salary = temp1;
84.       temp1 = low->empid;
85.       low->empid = pivot2;
86.       end->empid = temp1 ;
87.
88.    return prev;
89. }
90.
91. // Function to sort a linked list using Quick
```

```
92. // sort algorithm by swapping the next pointers
93. void BubSort ::  quicksort(emp *start,emp *end){
94.       // These are the variables to count swaps and comparisons.
95.     int swap = 0;
96.     int compare = 0;
97.     compare += 1;
98.      if(start == end){
99.            return;
100.           }
101.           emp* prev = partition(start,end);
102.           quicksort(start,prev);
103.
104.           compare += 1;
105.           if(prev != NULL && prev == start){
106.                 quicksort(prev->next,end);
107.           }else if(prev != NULL && prev->next != NULL){
108.                 quicksort(prev->next->next,end);
109.           }
110.
111.     }
112.
113.
114.     // Function to create an unsorted linked list
115.     emp* BubSort :: linkList_init() {
116.                  emp* head, *p;
117.                  int n;
118.                  cout<<endl<<" Enter the linked-list size:- ";
119.                       cin>>n;
120.                  for(int i = 0; i<n; i++){
121.                       if (i==0){
122.                             head = new emp;
123.                             p = head;
124.                             }else{
125.                                   p->next = new emp;
126.                                   p = p->next;
127.                             }
128.                             cout<<endl<<endl<<"\t Enter employee
   "<<i+1<<" id : ";
129.                             cin>>p->empid;
130.                             cout<<endl<<"\t Enter employee "<<i+1<<"
   salary($) : ";
131.                             cin>>p->salary;
132.                       }
133.                       p->next = NULL;
134.                       return head;
135.                 }
136.
137.     // Function for Bubble Sort.
```

```
138.      void BubSort :: bubble_sort(emp* head){
139.                      // These are the variables to count swaps and
    comparisons.
140.                      int swap = 0;
141.                      int compare = 0;
142.                          // It denotes the bubble node
143.                          emp* p = head;
144.                          emp* val = head;
145.                          // It denotes node previous to bubble node.
146.                          emp* q = NULL;
147.                          // It denotes node after the bubble node.
148.                          emp* r = NULL;
149.                          // Initial count considered 1.
150.                          int count = 1 ;
151.                          compare += 1;
152.                          while(val != NULL){
153.                                 r = val->next;
154.                                 p = val;
155.                                 q = head;
156.                                 compare += 1;
157.                                 // It will run until the node address after
    the bubble becomes NULL.
158.                                 while(r != NULL){
159.                                              compare += 1;
160.                                            // Checking if bubble salary is
    less than the salary in next node.
161.                                              if(p->salary < r->salary){
162.                                              q = p;
163.                                              p = r;
164.                                              }else if (p->salary == r-
    >salary && p->empid < r->empid){  // If the salaries are equal then compare
    on basis of employee id.
165.                                                     q = p;
166.                                                     p = r;
167.                                              }else{
168.                                                     swap += 1;
169.                                                     q->next = r;
170.                                                     p->next = r->next;
171.                                                     r->next = p;
172.                                                     q = r;
173.                                                     // If the swapping at
    head node occurs then we must take the address of head node.
174.                                                     if (count == 1 ){
175.                                                            compare += 1;
176.                                                            head = q;
177.                                                     }
178.                                              }
179.                                              r = p->next;
```

```
                                        count++;
                                }
                                val = val->next;
                        }

                        cout<<endl<<endl<<"\t The total number of swaps are:- "<<swap;
                        cout<<endl<<endl<<"\t The total number of comparisons are:- "<<compare;
                        cout<<endl<<endl<<"The sorted array is: "<<endl<<"\t";
                        // Calling a function to display sorted list.
                        SortedDisplay(head);
                }

        // Function to sort a linked list using selection
        // sort algorithm by swapping the next pointers
        void BubSort :: selection_sort(emp *head)
        {
            // These are the variables to count swaps and comparisons.
            int swap=0;
            int compare=0;
            compare += 1;
            for (emp* p = head; p->next != NULL; p = p->next){
                emp *min_idx = p;
                     compare += 1;
                for (emp* q = p->next; q != NULL; q = q->next){
                          compare += 1;
                     if (q->salary >= min_idx->salary){
                          if (q->salary == min_idx->salary){
                                if(q->empid > min_idx->empid){
                                       min_idx = q;
                                       }
                                }else{
                                       min_idx = q;
                                }
                          }
                     }
                compare += 1;
                     int temp = min_idx->salary;
                     min_idx->salary = p->salary;
                     p->salary = temp;
                     temp = min_idx->empid;
                     min_idx->empid = p->empid;
                     p->empid = temp;
              }
              cout<<endl<<endl<<"\t The total number of swaps are:- "<<swap;
```

```
224.            cout<<endl<<endl<<"\t The total number of comparisons are:-
    "<<compare;
225.            cout<<endl<<endl<<"\t The sorted list is:- ";
226.            display(head);
227.      }
228.
229.
230.      // Function to sort a linked list using insertion
231.      // sort algorithm by swapping the next pointers
232.      void BubSort :: insertion_sort(emp* head)
233.      {
234.            emp* val1 = head->next;
235.            // These are the variables to count swaps and comparisons.
236.            int swap = 0;
237.            int compare = 0;
238.
239.            while(val1!= NULL)
240.            {
241.                  compare += 1;
242.                  emp* val2 = head;
243.                  compare += 1;
244.                  // Checking the next greater value
245.                  while (val2 != val1)
246.                  {
247.                     compare += 1;
248.                     if (val1->salary > val2->salary)
249.                     {
250.                         int temp = val1->salary;
251.                         val1->salary=val2->salary;
252.                         val2->salary=temp;
253.                         temp = val1->empid;
254.                         val1->empid = val2->empid;
255.                         val2->empid = temp;
256.                         swap += 1;
257.                     }
258.                          else if(val1->salary == val2->salary && val1-
    >empid > val2->empid){
259.                            int temp = val1->salary;
260.                         val1->salary=val2->salary;
261.                         val2->salary=temp;
262.                         temp = val1->empid;
263.                         val1->empid = val2->empid;
264.                         val2->empid = temp;
265.                                     swap += 1;
266.                                }
267.                      val2=val2->next;
268.                  }
269.            val1 = val1->next;
```

```
270.            }
271.            cout<<endl<<endl<<"\t The total number of swaps are:- "<<swap;
272.            cout<<endl<<endl<<"\t The total number of comparisons are:-
    "<<compare;
273.            cout<<endl<<endl<<"\t The sorted list is:- ";
274.            display(head);
275.      }
276.
277.
278.      // Function to print the reverse sorted list.
279.      void BubSort :: SortedDisplay(emp* head){
280.            // Base case
281.          if (head == NULL)
282.          return;
283.
284.          // print the list after head node
285.          SortedDisplay(head->next);
286.
287.          // After everything else is printed, print head
288.          cout <<head->empid << " ";
289.      }
290.
291.      void BubSort :: display(emp* head){
292.                    emp *p ;
293.                    // Taking the value of head in p.
294.                    p = head;
295.                          cout<<endl<<endl<<"\t The employee Id's and
    corrosponding salaries are: "<<endl;
296.                          cout<<endl<<"\t ID's    "<<" Salary ";
297.                          // print the list after head node
298.                          while(p != NULL){
299.                                cout<<endl<<"\t "<<p->empid<<"\t "<<p-
    >salary<<" $";
300.                                p = p->next;
301.                          }
302.                  }
303.
304.      class first : BubSort
305.      {
306.            public:
307.                  // This function is calling the call function of
    BubSort.
308.                  first(){
309.                          cout<<endl<<endl<<endl<<"\t####################
    Sorting Algorithms  ########################"<<endl<<endl;
310.                          call();
311.                  }
312.      };
```

```
313.
314.        int main()
315.        {
316.                system("Color C0");
317.                first obj;
318.                return 0;
319.        }
```

❖ C++ Output:

```
      ###################### Sorting Algorithms  ########################

Enter the linked-list size:- 5

       Enter employee 1 id : 1

       Enter employee 1 salary($) : 200

       Enter employee 2 id : 4

       Enter employee 2 salary($) : 900

       Enter employee 3 id : 9

       Enter employee 3 salary($) : 500

       Enter employee 4 id : 6

       Enter employee 4 salary($) : 900

       Enter employee 5 id : 9

       Enter employee 5 salary($) : 800

      The employee Id's and corrosponding salaries are:

      ID's    Salary
      1       200 $
      4       900 $
      9       500 $
      6       900 $
      9       800 $

      The total number of swaps are:- 4

      The total number of comparisons are:- 19

      The sorted list is:-

      The employee Id's and corrosponding salaries are:

      ID's    Salary
      6       900 $
      4       900 $
      9       800 $
      9       500 $
      1       200 $
```

❖ Time Complexity Chart:

Algorithm Name	Best Case	Average Case	Worst Case
Bubble sort	Ω(n)	Ø(n2)	O(n2)
Selection sort	Ω(n2)	Ø(n2)	O(n2)
Insertion sort	Ω(n)	Ø(n2)	O(n2)
Merge sort	Ω(n×logn)	Ø(n×logn)	O(n×logn)
Quick sort	Ω(n×logn)	Ø(n×logn)	O(n2)

❖ Conclusion:

The following conclusion are drawn after doing the assignment:

1) **Selection sorts:** The simplest of sorting techniques. It's work very well for small files, also It's has a quite important application because each item is actually moved at most once. It has O (n2) time complexity, making it inefficient on large lists. Selection sort has one advantage over other sort techniques.

2) **Insertion sort:** It is very similar to selection sort. It is a simple sorting algorithm that builds the final sorted list one item at a time. It has O (n2) time complexity, it is much less efficient on large lists than more advanced algorithms such as quick sort or merge sort. However, insertion sort provides several advantages Simple implementation and, Efficient for small data sets.

3) **Merge sort:** It is a divide and conquer algorithm. It's Divide the list into two approximately equal sub lists, Then Sort the sub lists recursively. It has an O (n log n) Time complexity, merge sort is a stable sort, Merge sort is often the best choice for sorting a linked list.

4) **Quick sort:** In this sort an element called pivot is identified and that element is fixed in its place by moving all the elements less than that to its left and all the elements greater than that to its right. Since it partitions the element sequence into left, pivot and right it is referred as a sorting by partitioning.

5) **Bubble sort**: is a simple sorting algorithm that works by repeatedly, it's comparing each pair of adjacent items and swapping them if they are in the wrong order. This passing procedure is repeated until no swaps are required, indicating that the list is sorted. It has a O (n2) Time complexity means that its efficiency decreases dramatically on lists of more than a small number of elements.