Practical 1
Analyze running complexity of various sorting algorithm

GAHAN M. SARAIYA, 18MCEC10

18mcec10@nirmauni.ac.in

I. INTRODUCTION

Aim of this practical is to analyze the time complexity of sorting algorithms for various input size.

Algorithms analyzed are listed below:

- Bubble Sort

- Insertion Sort

- Selection Sort

- Quick Sort

- Merge Sort

- Heap Sort

II. IMPLEMENTATION

I.   Utility *utility.h*

```
1   //
2   // Created by jarvis on 17/8/18.
3   //
4
5   #ifndef DSA_LAB_UTILITY_H
6   #define DSA_LAB_UTILITY_H
7
8   #include <string.h>
9   #include <stdarg.h>
10
11  int write_log(const char *format, ...) {
12      if(DEBUG) {
13          va_list args;
14          va_start (args, format);
15          vprintf(format, args);
16          va_end (args);
```

```c
17          }
18      }
19
20      int *get_min_max(int *array, int no_of_elements, int min_max[]){
21          // get minimum and maximum of array
22      //     printf("elements of array: ");
23          for(int i=0; i<no_of_elements; i++){
24      //         printf("%d ", *(array + i));
25              if (*(array + i) < min_max[0])
26                  min_max[0] = *(array + i);
27              if (*(array + i) > min_max[1])
28                  min_max[1] = *(array + i);
29          }
30          return min_max;
31      }
32
33      int display_array(int *array, int no_of_elements){
34          // display given array of given size(no. of elements require because sizeof()
            ↪    returns max bound value)
35          write_log(": ");
36          for(int i=0; i<no_of_elements; i++){
37              write_log( "%d ", *(array + i));
38          }
39          return 0;
40      }
41
42
43      void swap(int *one, int *two){
44          // swap function to swap elements by location/address
45          int temp = *one;
46          *one = *two;
47          *two = temp;
48      }
49
50      char** split_string(char* str) {
51          // split string by separator space
52          char** splits = NULL;
53          char* token = strtok(str, " ");
54          int spaces = 0;
55
56          while (token) {
57              splits = realloc(splits, sizeof(char*) * ++spaces);
58              if (!splits) {
59                  return splits;
60              }
61              splits[spaces - 1] = token;
62              token = strtok(NULL, " ");
63          }
```

```
64      return splits;
65  }
66
67  void read_file_input() {
68      // under development function to read inputs from file
69      int ptr[100], count = 0, i, ar_count;
70      char c[100];
71      FILE *fp = fopen("file.in", "r");
72
73      char in = fgetc(fp);
74      // ar_count = (int) (in - '0');
75      printf("\narr\n");
76      while (in != EOF){
77          if ((int) (in -'0') == -16){
78              printf("\nspace\n");
79          }
80          else{
81              printf("%c - %d\n",in,  (int) (in - '0'));
82          }
83          in = fgetc(fp);
84      }
85      printf("\n\n");
86      fclose (fp);
87  }
88
89  #endif //DSA_LAB_UTILITY_H
```

II. Constants *constant.h*

```
1   //
2   // Created by jarvis on 17/8/18.
3   //
4
5   #ifndef DSA_LAB_CONSTANT_H
6   #define DSA_LAB_CONSTANT_H
7   #define TEST_NUM 5000
8   #define DEBUG 0
9
10  #endif //DSA_LAB_CONSTANT_H
```

III. Sorting Algorithms *sorting_algorithms.h*

```
1   //
2   // Created by jarvis on 17/8/18.
3   //
4
```

```c
#ifndef DSA_LAB_SORTING_ALGORITHMS_H
#define DSA_LAB_SORTING_ALGORITHMS_H

int* bubble_iterative(int* array, int start, int no_of_elements, int dummy){
    // Bubble sort algorithm
    // no_of_elements : parameter require to specify otherwise static size of
    ↪  array found with
    //   printf("\n--------------------Bubble sort--------------------\n");
    //   printf("\number of elements\t\tAlgorithm\t\n%d\t\t Bubble Sort",
    ↪ no_of_elements);
    register int i, j;
    bool flag;
    // size_t arr_size = sizeof(array);
    for (j=0; j < no_of_elements-1; j++){
        flag = false;
        for (i=0; i < no_of_elements-j-1; i++){
            if (array[i] > array[i+1]){
                swap(&array[i], &array[i+1]);
                flag = true;
            }
        }
        if (!flag)
            break;
    }
    return array;
}

int* insertion_iterative(int* array, int start, int no_of_elements, int dummy){
    // Insertion sort algorithm
    //   printf("\n--------------------Insertion sort--------------------\n");
    register int i, j, key;
    for (i = 1; i < no_of_elements; i++){
        key = array[i];
        j = i-1;
        while (j >= 0 && array[j] > key){
            array[j+1] = array[j];
            j = j-1;
        }
        array[j+1] = key;
    }
    return array;
}

int* selection_iterative(int* array, int start, int no_of_elements, int dummy){
    // Insertion sort algorithm
    //   printf("\n--------------------Selection sort--------------------\n");
    for (register int i = 0; i < no_of_elements-1; i++){  // iterate up to second
    ↪  last element
```

```
50          int min = i;   // set current index as minimum
51          for (register int j = i+1; j < no_of_elements; j++){   // iterate over all
    ↪    elements of certain range
52              if(*(array + j) < *(array + min))
53                  min = j;   // set new minimum index scanned/iterated so far
54          }
55          if (min != i)
56              swap(&array[i], &array[min]);   // swap minimum element
    ↪    scanned/iterated so far
57      }
58      return array;
59  }
60
61  int* quick_recursive(int *array, int start, int no_of_elements, int dummy) {
62      no_of_elements = no_of_elements-1;
63      int partition (int* array, int start, int no_of_elements) {
64          int pivot = array[no_of_elements];    // select pivot
65          int i = (start - 1);   // get index of smaller element
66
67          for (int j=start; j <= no_of_elements- 1; j++) {
68              if (array[j] <= pivot) {
69                  i++;     // increment index of smaller element
70                  swap(&array[i], &array[j]);
71              }
72          }
73          swap(&array[i + 1], &array[no_of_elements]);
74          return (i + 1);
75      }
76      // PROCESSING
77      if (start < no_of_elements) {
78          int partition_index = partition(array, start, no_of_elements);
79          quick_recursive(array, start, partition_index - 1, dummy);
80          quick_recursive(array, partition_index + 1, no_of_elements, dummy);
81      }
82      return array;
83  }
84
85  int* merge_recursive(int* array, int low, int high, int no_of_elements){
86      // low: left start node
87      // high: right end node |i.e. (number of elements - 1)
88
89      void conquer(int array[], int low, int mid, int high, int no_of_elements){
90          int temp[no_of_elements];
91          int num1, num2, i=low, j=mid+1, k=0;
92          while(i<=mid && j<=high)
93          {
94              if(array[i] < array[j])
95                  temp[k++] = array[i++];
```

```
 96              else
 97                  temp[k++] = array[j++];
 98          }
 99          while(i<=mid)
100              temp[k++] = array[i++];
101
102          while(j<=high)
103              temp[k++] = array[j++];
104
105          //Transfer elements from temp[] back to a[]
106          for(i=low,j=0;i<=high;i++,j++)
107              array[i]=temp[j];
108      }
109      // PROCESSING
110      int mid;
111      if (low < high){
112          mid = (low + high) / 2;
113          merge_recursive(array, low, mid, no_of_elements);
114          merge_recursive(array, mid+1, high, no_of_elements);
115
116          conquer(array, low, mid, high, no_of_elements);
117      }
118      return array;
119  }
120
121  int* heap_recursive(int* array, int start, int no_of_elements, int dummy){
122      int* heapify(int* array, int heap_size, int idx){
123          // heapify array == rearrange array to follow heap structure/rule
124          // heap_size: no_of_elements
125          int root = idx;  // consider given node as current possible root node
126          int left = 2*idx + 1;
127          int right = 2*idx + 2;
128
129          if (left < heap_size && *(array + left) > *(array + root)){  // right
             ↪  child > eligible root/largest
130              root = left;
131          }
132          if (right < heap_size && *(array + right) > *(array + root)){  // right
             ↪  child > eligible root/largest
133              root = right;
134          }
135          if (root != idx){  // root node is not largest
136              swap(&array[idx], &array[root]);
137              heapify(array, heap_size, idx);
138          }
139      }
140      // PROCESSING
141      for (int i=(no_of_elements/2)-1; i >= 0; i--){
```

```
142        heapify(array, no_of_elements, i);
143    }
144    for (int i=no_of_elements-1; i >=0; i--){
145        // move current root to end
146        swap(&array[0], &array[i]);
147        heapify(array, i, 0);  // max heapify
148    }
149    return array;
150 }
151
152 #endif //DSA_LAB_SORTING_ALGORITHMS_H
```

IV.   Main Program - *program.c*

```
1  /*
2   sorting algorithm analysis
3   algorithms:
4   graph: (total number v/s total time)
5        sorted
6        reverse sorted
7        unsorted
8     logarithmic scale (for variation in density)
9   */
10 #include <stdio.h>
11 #include <stdbool.h>
12 #include <string.h>
13 #include <stdlib.h>
14 #include <time.h>
15 #include "constant.h"
16 #include "utility.h"
17 #include "sorting_algorithms.h"
18
19 //int TEST_NUM = 10;
20
21
22 int* generate_array(int max_element, int sort_flag){
23     // generate array of n elements
24
25 //    static int* array;
26 //    array = (int*)malloc(TEST_NUM * sizeof(int));
27     static int array[TEST_NUM];
28     if (sort_flag == 0) {  // generate random unsorted numbers if flag is 0
29         for (register int i = 0; i < max_element; i++) {
30             array[i] = rand();
31         }
32     }
33     else if (sort_flag == 1){
```

```
34          for(int i=0; i < max_element; i++){  // generate sorted "ascending"
         ↪   numbers if flag is true
35              array[i] = i;
36          }
37      }
38      else if (sort_flag == 2){
39          for(int i=max_element; i > 0; i--){  // generate sorted "descending"
         ↪   numbers if flag is true
40              array[max_element-i] = i;
41          }
42      }
43      else{
44          for(int i=0; i < max_element; i++){  // all elements same
45              array[i] = sort_flag;
46          }
47      }
48      return array;
49  }
50
51  void analysis(int* (*f)(int *, int,  int, int), char algo_name[]){
52      int *arr_ptr;
53      FILE *fptr = fopen("analysis.csv", "a");
54      clock_t t;
55      double cpu_time_consumption;
56      int number = TEST_NUM;
57      if (algo_name == "Merge sort (Recursive)")
58          number = TEST_NUM - 1;
59      printf("\nAnalysis of %s", algo_name);
60
61      // unsorted elements
62      arr_ptr = generate_array(number, 0);
63      display_array(arr_ptr, number);
64      printf("\n- for unsorted %d elements: ", TEST_NUM);
65      t = clock();
66      (*f)(arr_ptr, 0, number, number);
67      t = clock() - t;
68      cpu_time_consumption = ((double) (t)) / CLOCKS_PER_SEC;
69      printf(":: %f", cpu_time_consumption);
70      display_array(arr_ptr, number);
71      fprintf(fptr, "%d,%f,%s,random\n", number, cpu_time_consumption, algo_name);
72
73      // ascending sorted
74      printf("\n- for sorted (Ascending) %d elements: ", TEST_NUM);
75      arr_ptr = generate_array(TEST_NUM, 1);
76      display_array(arr_ptr, TEST_NUM);
77      t = clock();
78      (*f)(arr_ptr, 0, number, number);
79      t = clock() - t;
```

```
80      cpu_time_consumption = ((double) (t)) / CLOCKS_PER_SEC;
81      printf(":: %f\n", cpu_time_consumption);
82      display_array(arr_ptr, TEST_NUM);
83      fprintf(fptr, "%d,%f,%s,ascending\n", number, cpu_time_consumption,
    ↪   algo_name);
84
85      // descending sorted
86      printf("\n- for sorted (Descending) %d elements: ", TEST_NUM);
87      arr_ptr = generate_array(TEST_NUM, 2);
88      display_array(arr_ptr, TEST_NUM);
89      t = clock();
90      (*f)(arr_ptr, 0, number, number);
91      t = clock() - t;
92      cpu_time_consumption = ((double) (t)) / CLOCKS_PER_SEC;
93      printf(":: %f\n", cpu_time_consumption);
94      display_array(arr_ptr, TEST_NUM);
95      fprintf(fptr, "%d,%f,%s,descending\n", number, cpu_time_consumption,
    ↪   algo_name);
96
97      // all elements equal
98      printf("\n- for all equal %d elements: ", TEST_NUM);
99      arr_ptr = generate_array(TEST_NUM, 500);
100     display_array(arr_ptr, TEST_NUM);
101     t = clock();
102     (*f)(arr_ptr, 0, number, number);
103     t = clock() - t;
104     cpu_time_consumption = ((double) (t)) / CLOCKS_PER_SEC;
105     printf(":: %f\n", cpu_time_consumption);
106     display_array(arr_ptr, TEST_NUM);
107     fprintf(fptr, "%d,%f,%s,same\n", number, cpu_time_consumption, algo_name);
108     fclose(fptr);
109 }
110
111 int main(){
112     int i, *res, *arr_ptr;
113     clock_t t;
114     double cpu_time_consumption;
115     printf("Initializing Sorting Algorithm for %d numbers...\n", TEST_NUM);
116     analysis(bubble_iterative, "Bubble_sort_(Iterative)");
117     analysis(insertion_iterative, "Insertion_sort_(Iterative)");
118     analysis(selection_iterative, "Selection_sort_(Iterative)");
119     analysis(quick_recursive, "Quick_sort_(Recursive)");
120     analysis(merge_recursive, "Merge_sort_(Recursive)");
121     analysis(heap_recursive, "Heap_sort_(Recursive)");
122     // read_file_input();
123     printf("\n\n-*-*-*-*-*-*-*-*-*-END OF PROGRAM*-*-*-*-*-*-*-*-*-*-*-*-*\n\n");
124     return 0;
125 }
```

```
126
127   /*
128   Initializing Sorting Algorithm for 10,00,000 numbers...
129
130   Analysis of Insertion sort (Iterative)
131   - for unsorted 1000000 elements: 374.238000
132   */
```

III. ANALYSIS

All mentioned algorithms are tested for input length: 10, 10^1, 10^2, 10^3, 10^4, 10^5 with various array construction listed below:

- All elements are in ascending order

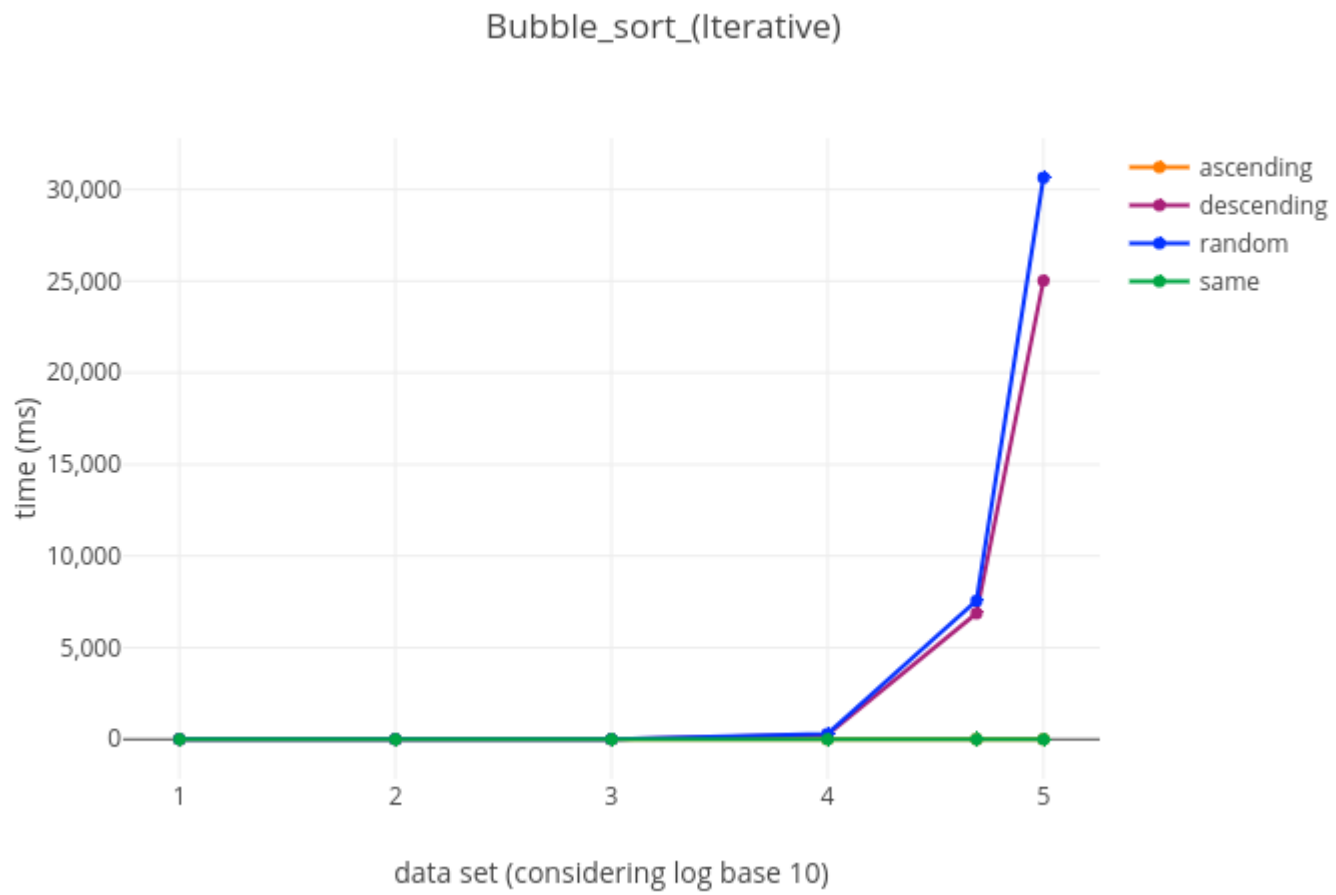- All elements are in descending order

- All elements are in random order

- All elements are same

I. Bubble Sort

Figure 1: Bubble Sort

II. Insertion Sort
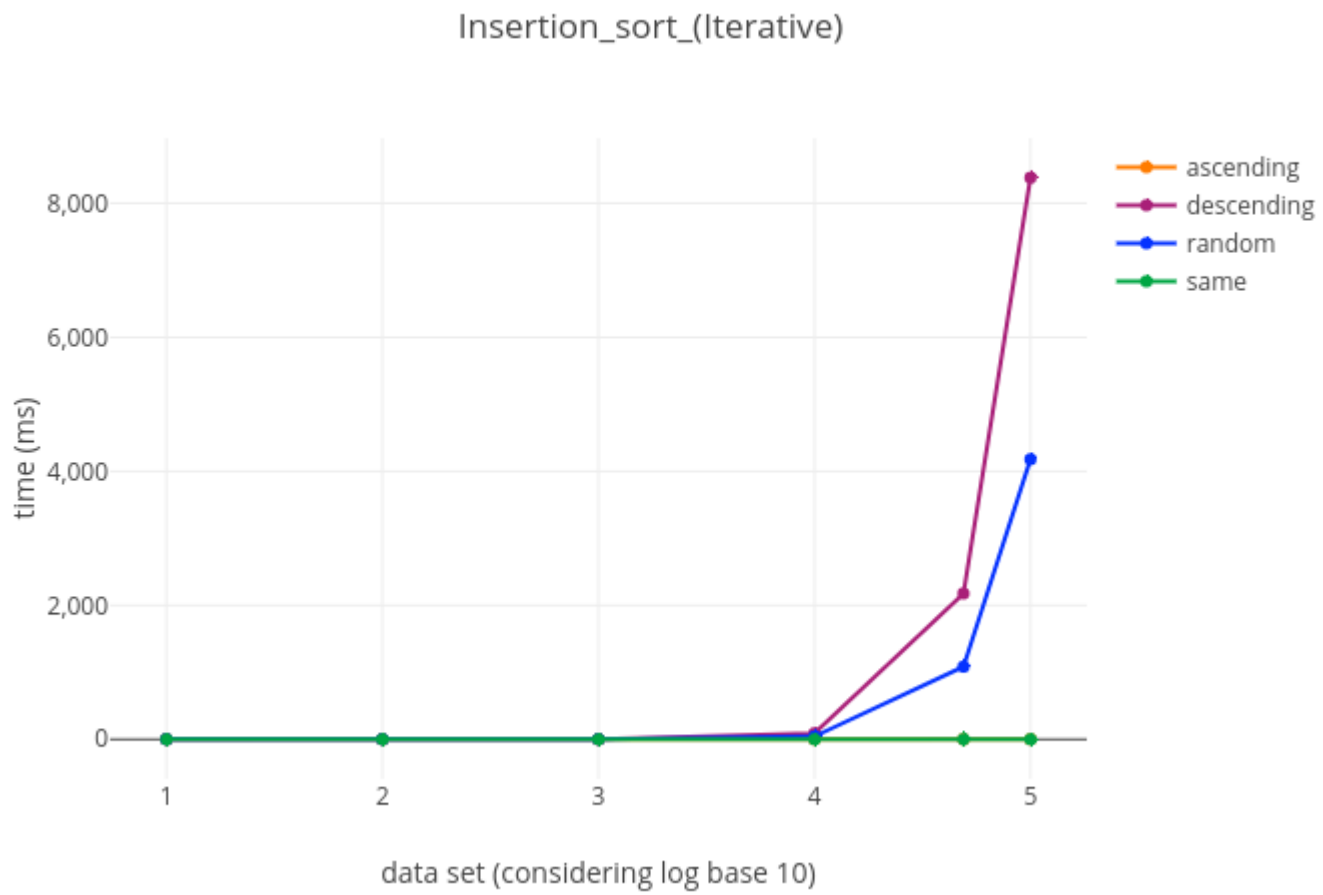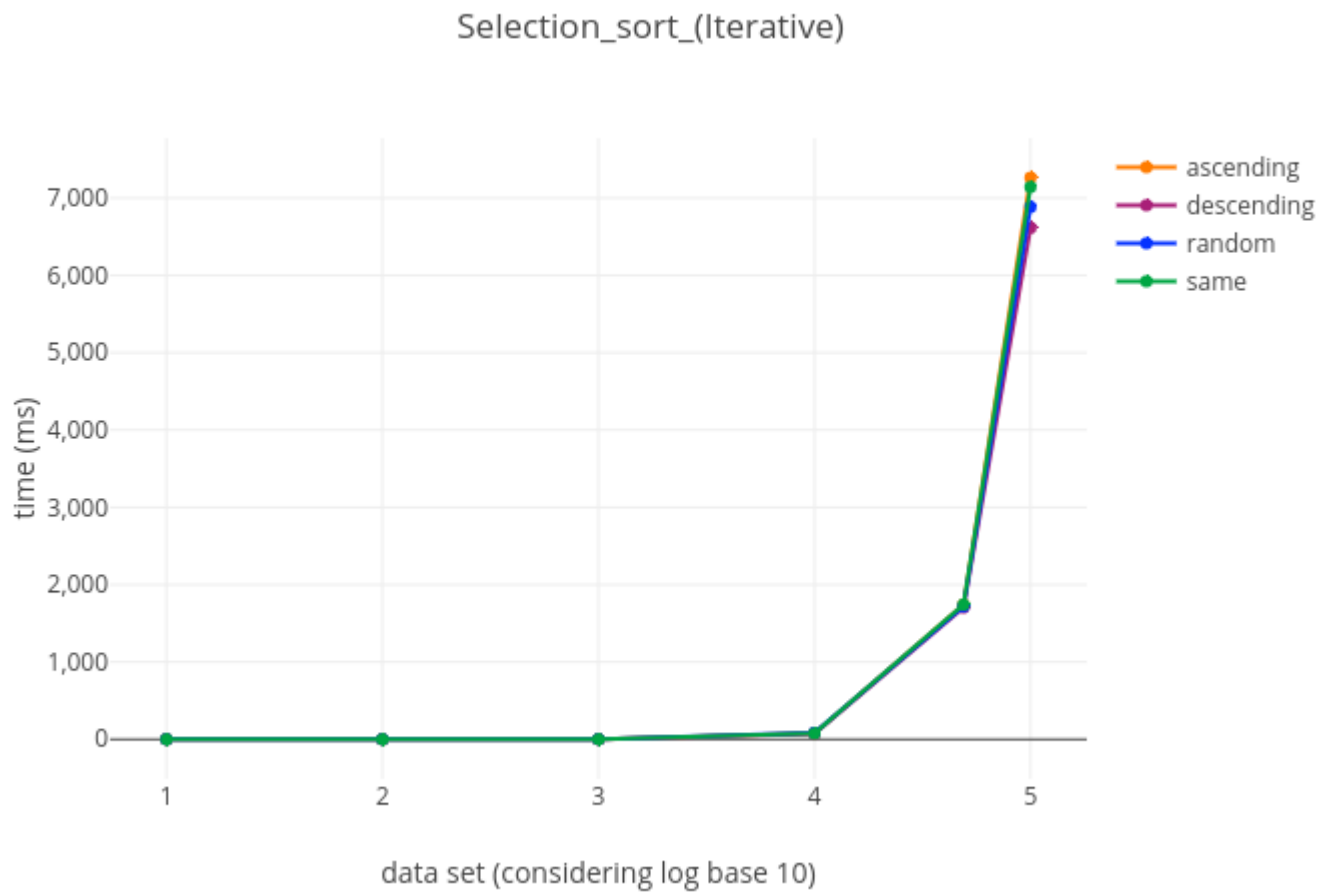

Figure 2: Insertion Sort

III. Selection Sort

Selection_sort_(Iterative)

Figure 3: Selection Sort

IV. Quick Sort

Quick_sort_(Recursive)
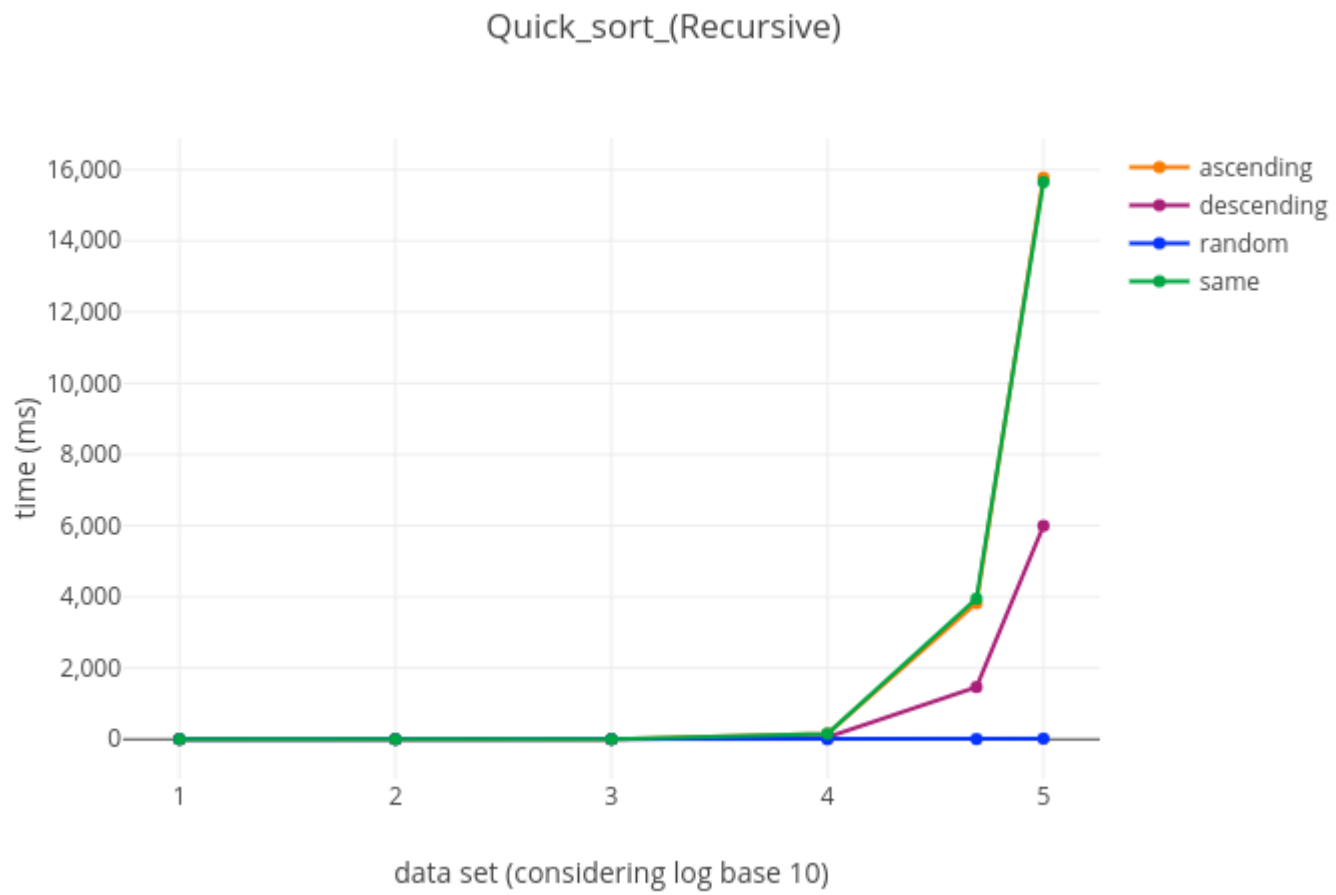
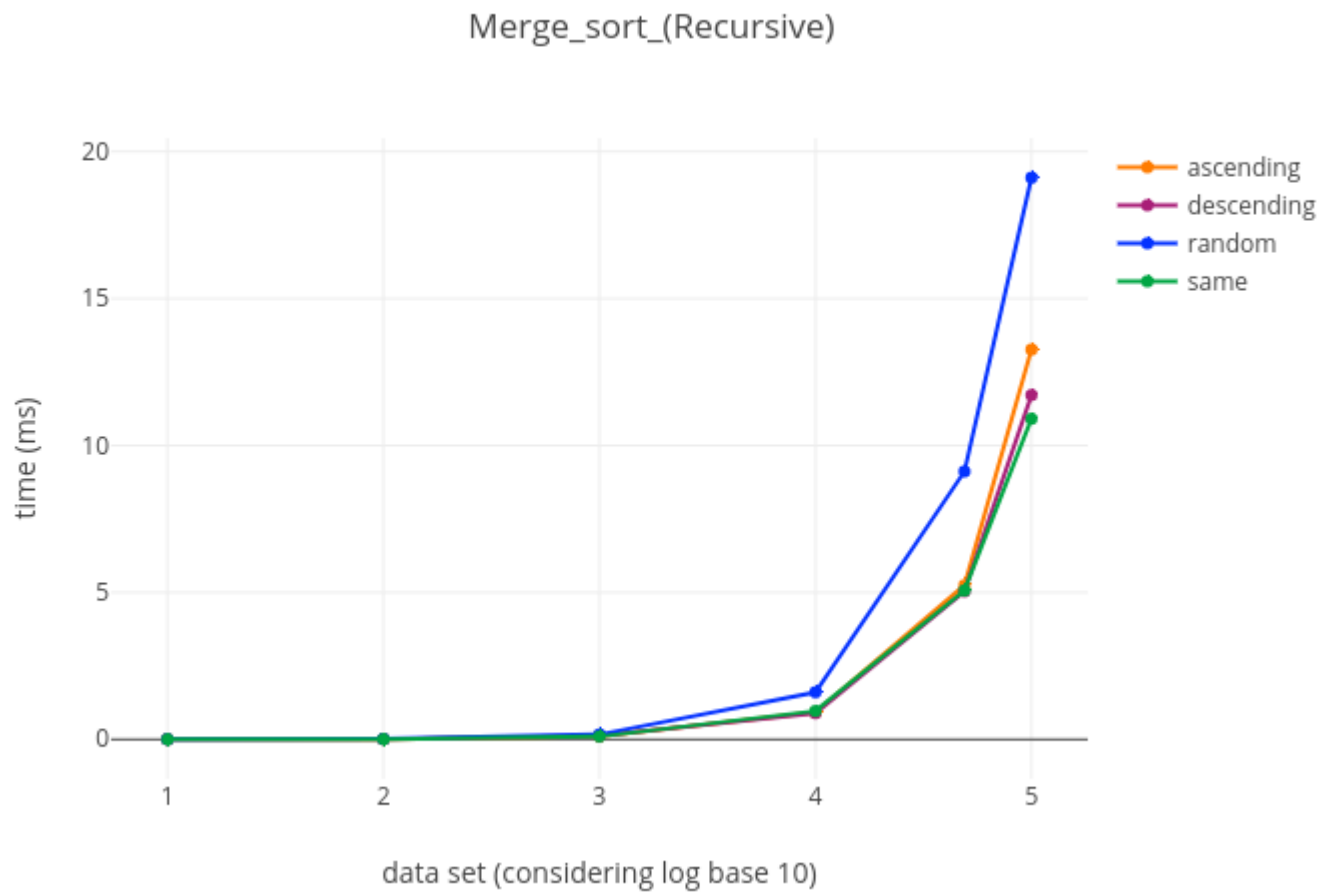

Figure 4: Quick Sort

V. Merge Sort

Merge_sort_(Recursive)

Figure 5: Merge Sort

VI. Heap Sort

Figure 6: Heap Sort