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*

```c
//
// Created by jarvis on 17/8/18.
//

#ifndef DSA_LAB_UTILITY_H
#define DSA_LAB_UTILITY_H

#include <string.h>
#include <stdarg.h>

int write_log(const char *format, ...) {
    if(DEBUG) {
        va_list args;
        va_start (args, format);
        vprintf(format, args);
        va_end (args);
```

```
    }
}

int *get_min_max(int *array, int no_of_elements, int min_max[]){
    // get minimum and maximum of array
//    printf("elements of array: ");
    for(int i=0; i<no_of_elements; i++){
//        printf("%d ", *(array + i));
        if (*(array + i) < min_max[0])
            min_max[0] = *(array + i);
        if (*(array + i) > min_max[1])
            min_max[1] = *(array + i);
    }
    return min_max;
}

int display_array(int *array, int no_of_elements){
    // display given array of given size(no. of elements require because sizeof()
     ↪  returns max bound value)
    write_log(": ");
    for(int i=0; i<no_of_elements; i++){
        write_log( "%d ", *(array + i));
    }
    return 0;
}


void swap(int *one, int *two){
    // swap function to swap elements by location/address
    int temp = *one;
    *one = *two;
    *two = temp;
}

char** split_string(char* str) {
    // split string by separator space
    char** splits = NULL;
    char* token = strtok(str, " ");
    int spaces = 0;

    while (token) {
        splits = realloc(splits, sizeof(char*) * ++spaces);
        if (!splits) {
            return splits;
        }
        splits[spaces - 1] = token;
        token = strtok(NULL, " ");
    }
```

```
    return splits;
}

void read_file_input() {
    // under development function to read inputs from file
    int ptr[100], count = 0, i, ar_count;
    char c[100];
    FILE *fp = fopen("file.in", "r");

    char in = fgetc(fp);
    // ar_count = (int) (in - '0');
    printf("\narr\n");
    while (in != EOF){
        if ((int) (in -'0') == -16){
            printf("\nspace\n");
        }
        else{
            printf("%c - %d\n",in,  (int) (in - '0'));
        }
        in = fgetc(fp);
    }
    printf("\n\n");
    fclose (fp);
}

#endif //DSA_LAB_UTILITY_H
```

II. Constants *constant.h*

```
//
// Created by jarvis on 17/8/18.
//

#ifndef DSA_LAB_CONSTANT_H
#define DSA_LAB_CONSTANT_H
#define TEST_NUM 5000
#define DEBUG 0

#endif //DSA_LAB_CONSTANT_H
```

III. Sorting Algorithms *sorting_algorithms.h*

```
//
// Created by jarvis on 17/8/18.
//

```

```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
```

```
        int min = i;  // set current index as minimum
        for (register int j = i+1; j < no_of_elements; j++){  // iterate over all elements of certain range
            if(*(array + j) < *(array + min))
                min = j;  // set new minimum index scanned/iterated so far
        }
        if (min != i)
            swap(&array[i], &array[min]);  // swap minimum element scanned/iterated so far
    }
    return array;
}

int* quick_recursive(int *array, int start, int no_of_elements, int dummy) {
    no_of_elements = no_of_elements-1;
    int partition (int* array, int start, int no_of_elements) {
        int pivot = array[no_of_elements];    // select pivot
        int i = (start - 1);  // get index of smaller element

        for (int j=start; j <= no_of_elements- 1; j++) {
            if (array[j] <= pivot) {
                i++;    // increment index of smaller element
                swap(&array[i], &array[j]);
            }
        }
        swap(&array[i + 1], &array[no_of_elements]);
        return (i + 1);
    }
    // PROCESSING
    if (start < no_of_elements) {
        int partition_index = partition(array, start, no_of_elements);
        quick_recursive(array, start, partition_index - 1, dummy);
        quick_recursive(array, partition_index + 1, no_of_elements, dummy);
    }
    return array;
}

int* merge_recursive(int* array, int low, int high, int no_of_elements){
    // low: left start node
    // high: right end node |i.e. (number of elements - 1)

    void conquer(int array[], int low, int mid, int high, int no_of_elements){
        int temp[no_of_elements];
        int num1, num2, i=low, j=mid+1, k=0;
        while(i<=mid && j<=high)
        {
            if(array[i] < array[j])
                temp[k++] = array[i++];
```

```
            else
                temp[k++] = array[j++];
        }
        while(i<=mid)
            temp[k++] = array[i++];

        while(j<=high)
            temp[k++] = array[j++];

        //Transfer elements from temp[] back to a[]
        for(i=low,j=0;i<=high;i++,j++)
            array[i]=temp[j];
    }
    // PROCESSING
    int mid;
    if (low < high){
        mid = (low + high) / 2;
        merge_recursive(array, low, mid, no_of_elements);
        merge_recursive(array, mid+1, high, no_of_elements);

        conquer(array, low, mid, high, no_of_elements);
    }
    return array;
}

int* heap_recursive(int* array, int start, int no_of_elements, int dummy){
    int* heapify(int* array, int heap_size, int idx){
        // heapify array == rearrange array to follow heap structure/rule
        // heap_size: no_of_elements
        int root = idx;  // consider given node as current possible root node
        int left = 2*idx + 1;
        int right = 2*idx + 2;

        if (left < heap_size && *(array + left) > *(array + root)){  // right child > eligible root/largest
            root = left;
        }
        if (right < heap_size && *(array + right) > *(array + root)){  // right child > eligible root/largest
            root = right;
        }
        if (root != idx){  // root node is not largest
            swap(&array[idx], &array[root]);
            heapify(array, heap_size, idx);
        }
    }
    // PROCESSING
    for (int i=(no_of_elements/2)-1; i >= 0; i--){
```

```c
        heapify(array, no_of_elements, i);
    }
    for (int i=no_of_elements-1; i >=0; i--){
        // move current root to end
        swap(&array[0], &array[i]);
        heapify(array, i, 0);  // max heapify
    }
    return array;
}

#endif //DSA_LAB_SORTING_ALGORITHMS_H
```

IV. Main Program - *program.c*

```c
/*
 sorting algorithm analysis
 algorithms:
 graph: (total number v/s total time)
        sorted
        reverse sorted
        unsorted
    logarithmic scale (for variation in density)
 */
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "constant.h"
#include "utility.h"
#include "sorting_algorithms.h"

//int TEST_NUM = 10;


int* generate_array(int max_element, int sort_flag){
    // generate array of n elements

//    static int* array;
//    array = (int*)malloc(TEST_NUM * sizeof(int));
    static int array[TEST_NUM];
    if (sort_flag == 0) {  // generate random unsorted numbers if flag is 0
        for (register int i = 0; i < max_element; i++) {
            array[i] = rand();
        }
    }
    else if (sort_flag == 1){
```

```c
        for(int i=0; i < max_element; i++){  // generate sorted "ascending"
         ↪ numbers if flag is true
            array[i] = i;
        }
    }
    else if (sort_flag == 2){
        for(int i=max_element; i > 0; i--){  // generate sorted "descending"
         ↪ numbers if flag is true
            array[max_element-i] = i;
        }
    }
    else{
        for(int i=0; i < max_element; i++){  // all elements same
            array[i] = sort_flag;
        }
    }
    return array;
}

void analysis(int* (*f)(int *, int,  int, int), char algo_name[]){
    int *arr_ptr;
    FILE *fptr = fopen("analysis.csv", "a");
    clock_t t;
    double cpu_time_consumption;
    int number = TEST_NUM;
    if (algo_name == "Merge sort (Recursive)")
        number = TEST_NUM - 1;
    printf("\nAnalysis of %s", algo_name);

    // unsorted elements
    arr_ptr = generate_array(number, 0);
    display_array(arr_ptr, number);
    printf("\n- for unsorted %d elements: ", TEST_NUM);
    t = clock();
    (*f)(arr_ptr, 0, number, number);
    t = clock() - t;
    cpu_time_consumption = ((double) (t)) / CLOCKS_PER_SEC;
    printf(":: %f", cpu_time_consumption);
    display_array(arr_ptr, number);
    fprintf(fptr, "%d,%f,%s,random\n", number, cpu_time_consumption, algo_name);

    // ascending sorted
    printf("\n- for sorted (Ascending) %d elements: ", TEST_NUM);
    arr_ptr = generate_array(TEST_NUM, 1);
    display_array(arr_ptr, TEST_NUM);
    t = clock();
    (*f)(arr_ptr, 0, number, number);
    t = clock() - t;
```

```
    cpu_time_consumption = ((double) (t)) / CLOCKS_PER_SEC;
    printf(":: %f\n", cpu_time_consumption);
    display_array(arr_ptr, TEST_NUM);
    fprintf(fptr, "%d,%f,%s,ascending\n", number, cpu_time_consumption, algo_name);

    // descending sorted
    printf("\n- for sorted (Descending) %d elements: ", TEST_NUM);
    arr_ptr = generate_array(TEST_NUM, 2);
    display_array(arr_ptr, TEST_NUM);
    t = clock();
    (*f)(arr_ptr, 0, number, number);
    t = clock() - t;
    cpu_time_consumption = ((double) (t)) / CLOCKS_PER_SEC;
    printf(":: %f\n", cpu_time_consumption);
    display_array(arr_ptr, TEST_NUM);
    fprintf(fptr, "%d,%f,%s,descending\n", number, cpu_time_consumption, algo_name);

    // all elements equal
    printf("\n- for all equal %d elements: ", TEST_NUM);
    arr_ptr = generate_array(TEST_NUM, 500);
    display_array(arr_ptr, TEST_NUM);
    t = clock();
    (*f)(arr_ptr, 0, number, number);
    t = clock() - t;
    cpu_time_consumption = ((double) (t)) / CLOCKS_PER_SEC;
    printf(":: %f\n", cpu_time_consumption);
    display_array(arr_ptr, TEST_NUM);
    fprintf(fptr, "%d,%f,%s,same\n", number, cpu_time_consumption, algo_name);
    fclose(fptr);
}

int main(){
    int i, *res, *arr_ptr;
    clock_t t;
    double cpu_time_consumption;
    printf("Initializing Sorting Algorithm for %d numbers...\n", TEST_NUM);
    analysis(bubble_iterative, "Bubble_sort_(Iterative)");
    analysis(insertion_iterative, "Insertion_sort_(Iterative)");
    analysis(selection_iterative, "Selection_sort_(Iterative)");
    analysis(quick_recursive, "Quick_sort_(Recursive)");
    analysis(merge_recursive, "Merge_sort_(Recursive)");
    analysis(heap_recursive, "Heap_sort_(Recursive)");
    // read_file_input();
    printf("\n\n-*-*-*-*-*-*-*-*-*-END OF PROGRAM*-*-*-*-*-*-*-*-*-*-*-*-*-*\n\n");
    return 0;
}
```

```
/*
Initializing Sorting Algorithm for 10,00,000 numbers...

Analysis of Insertion sort (Iterative)
- for unsorted 1000000 elements: 374.238000
*/
```

III. Analysis

All mentioned algorithms are tested for input length: 10, 10^1, 10^2, 10^3, 10^4, 10^5

I. Bubble Sort

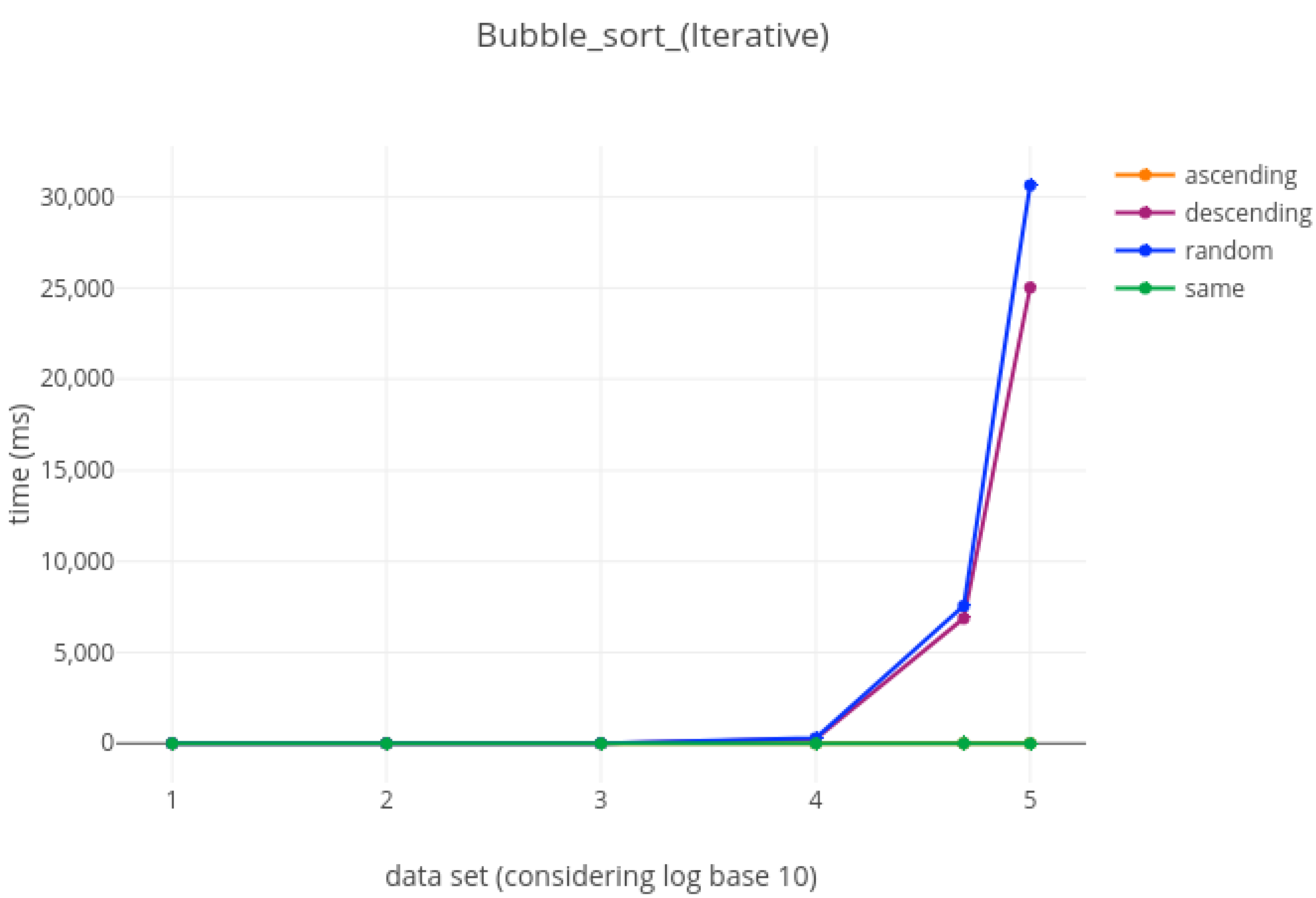

Figure 1: Bubble Sort

II. Insertion Sort

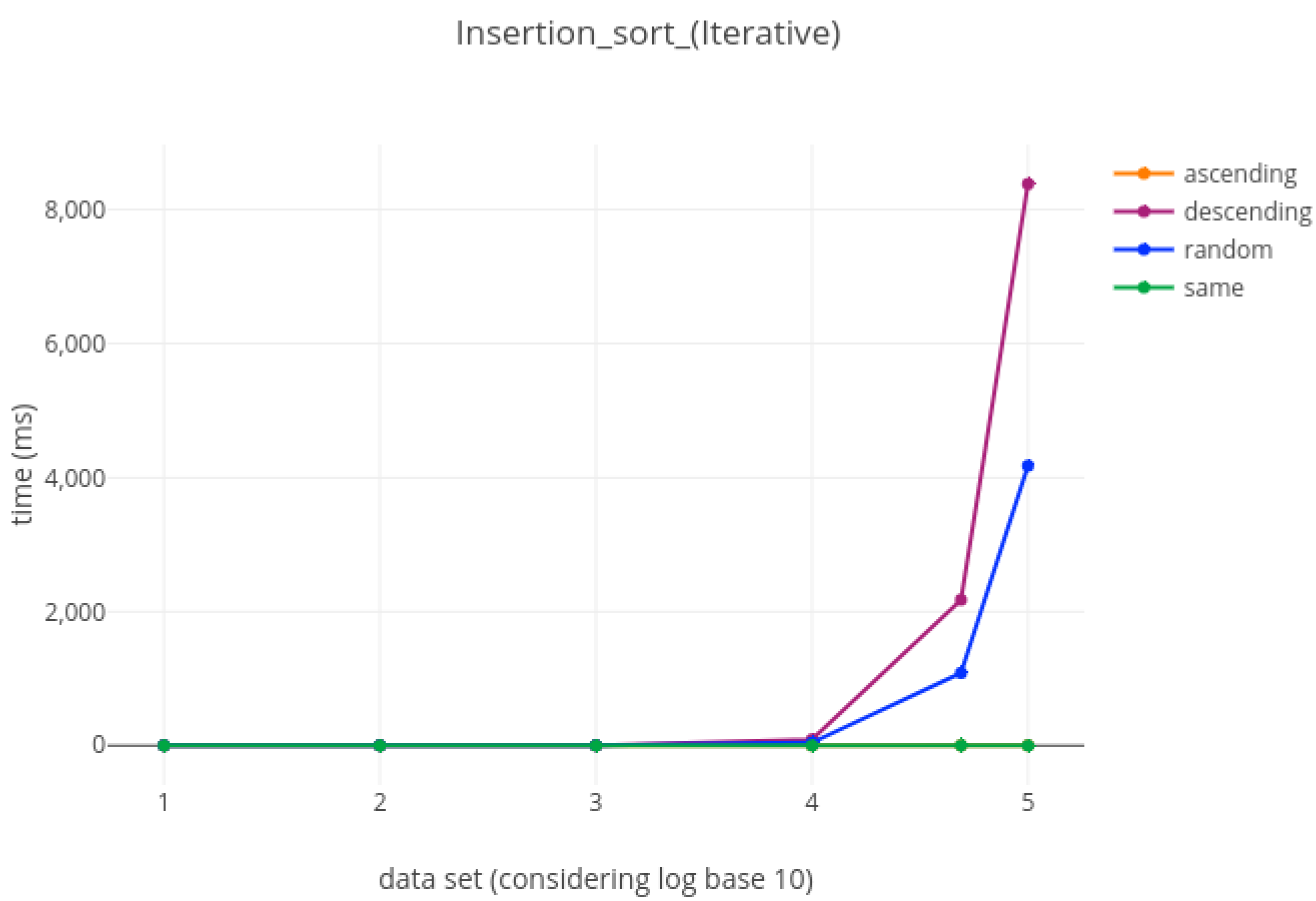

Figure 2: Insertion Sort

III. Selection Sort

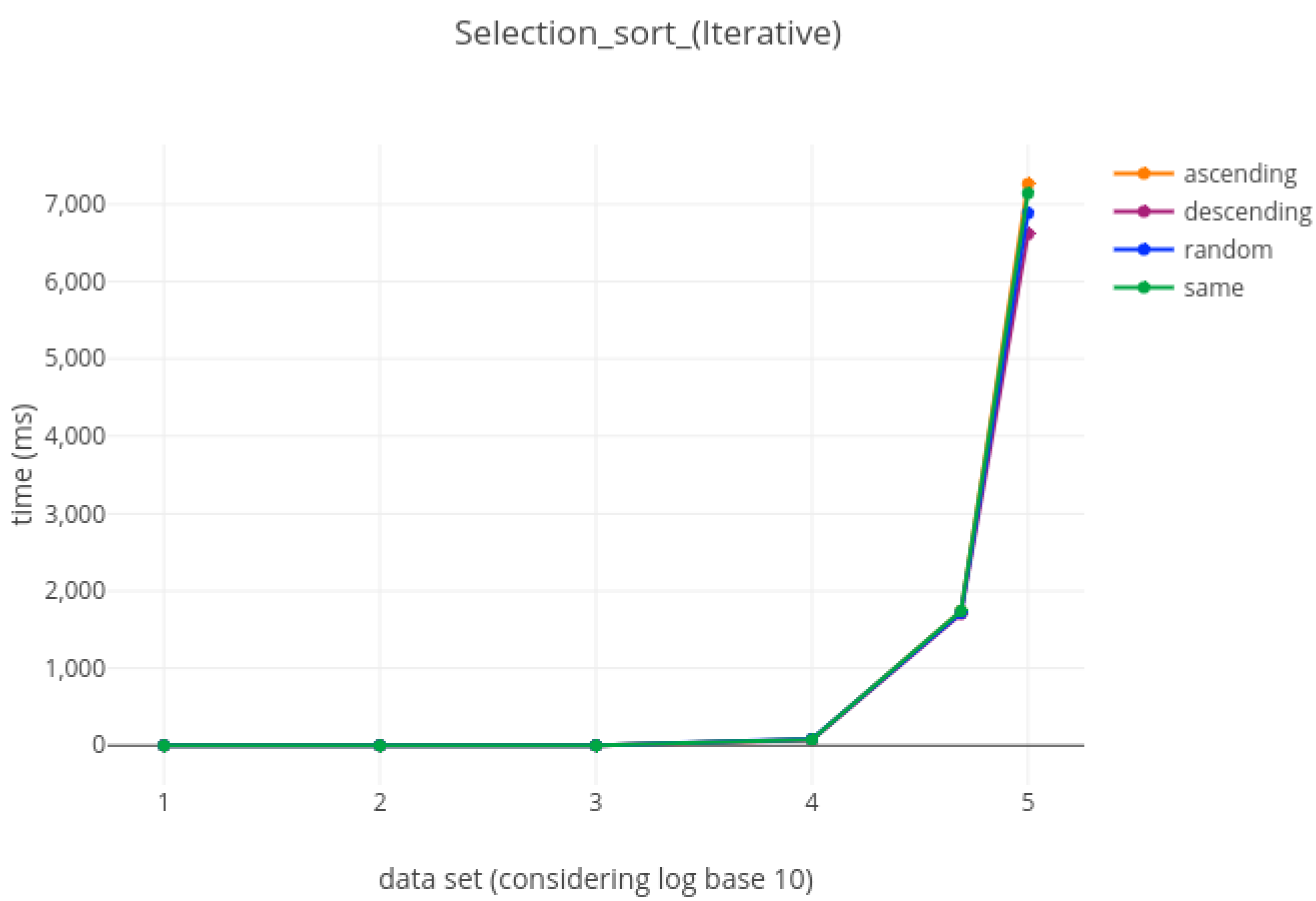

Figure 3: Selection Sort

IV. Quick Sort

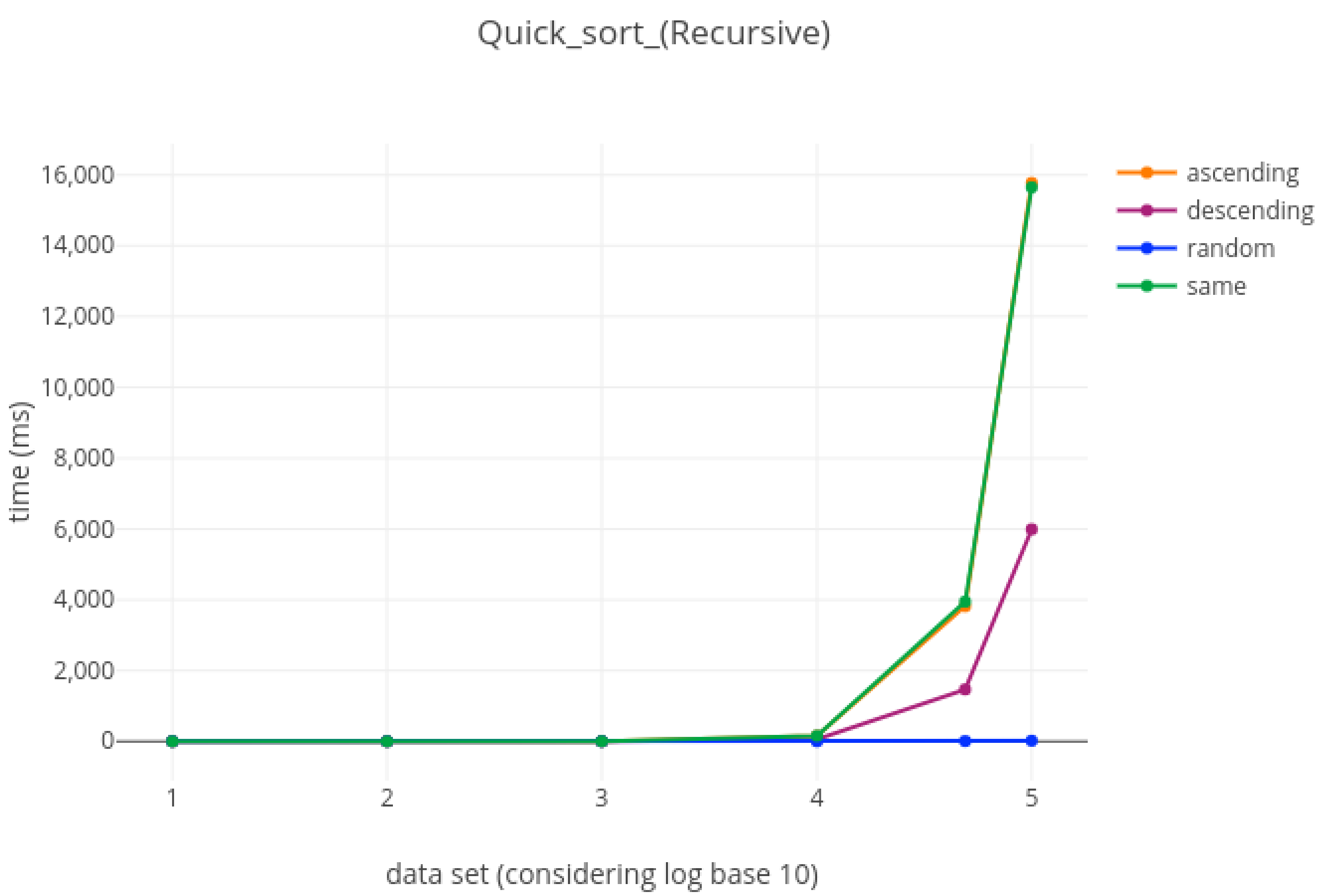

Figure 4: Quick Sort

V. Merge Sort

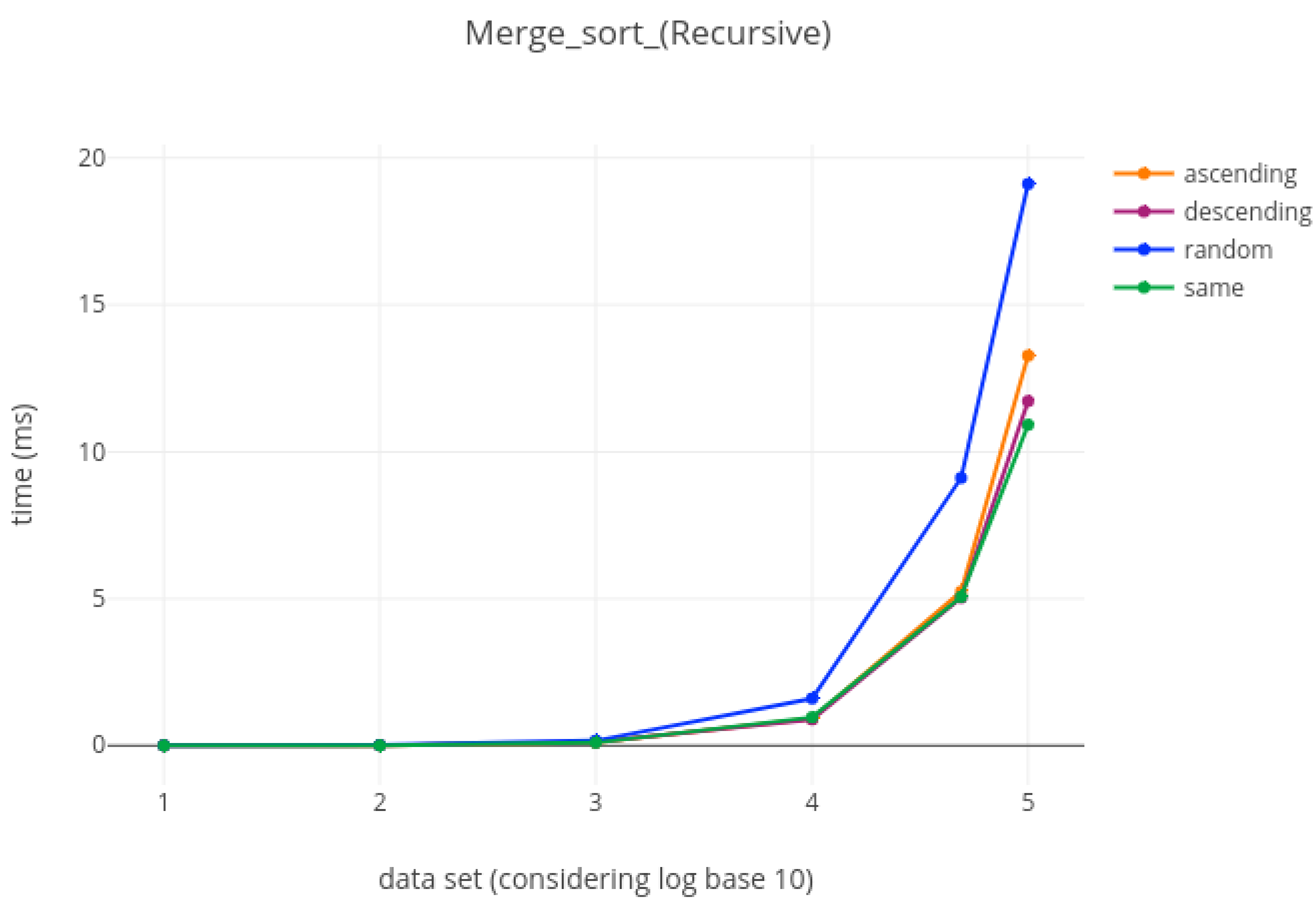

Figure 5: Merge Sort

VI. Heap Sort

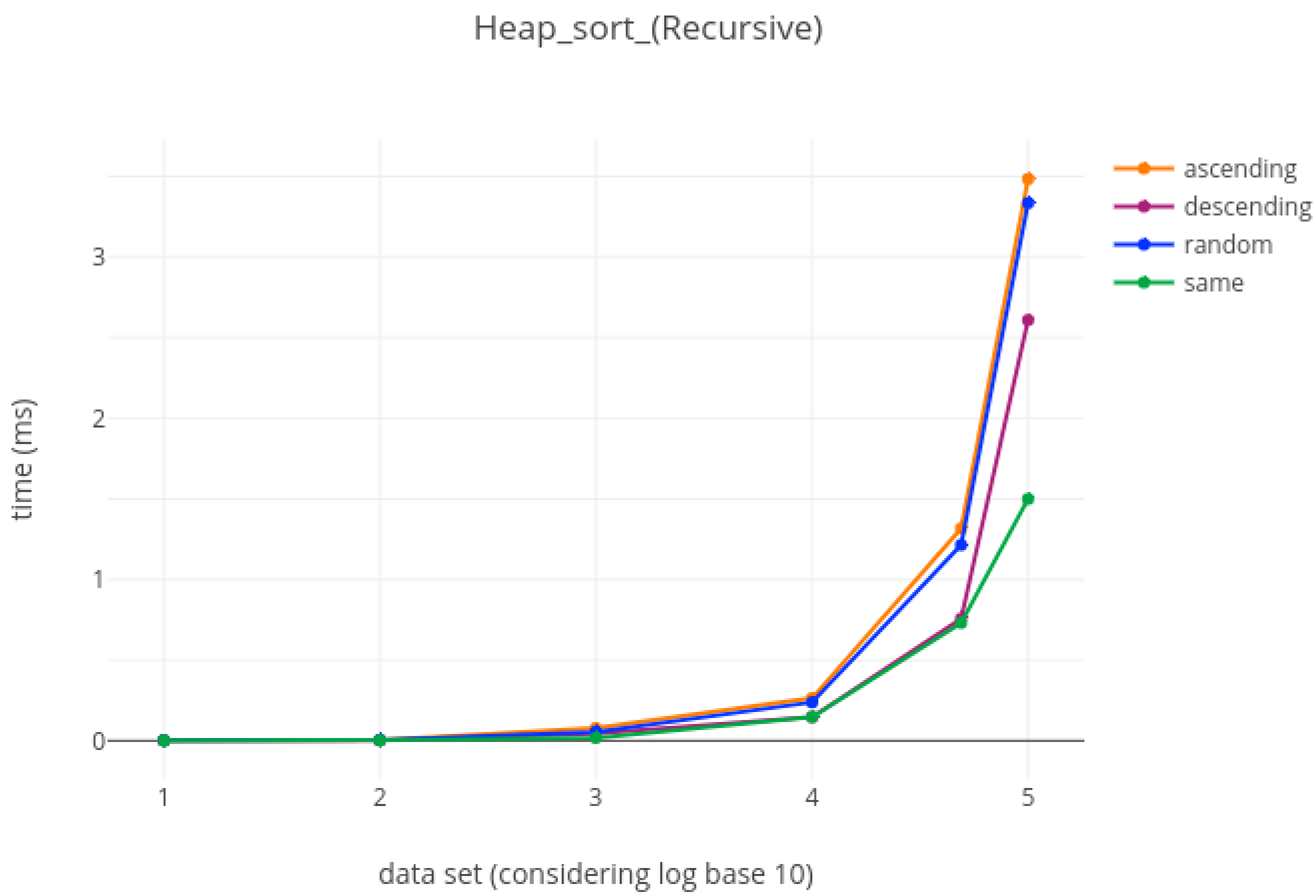

Figure 6: Heap Sort

IV. Summary

✓ If directory fits in memory then point query requires only 1 disk access

✓ Empty buckets can be merge with it's split image when directory becomes half of size